

# FORECASTING AND PREDICTIVE MODELING

## LECTURE 2

Lect. PhD. Oneț-Marian Zsuzsanna

Babeș - Bolyai University  
Computer Science and Mathematics Faculty

2024 - 2025

- Course organization
- What is forecasting?
- Types of time series
- Our forecasting workflow

- Introduction to R
  - Packages
  - Data in R
  - Visualizations

- R is an elegant and comprehensive statistical and graphical programming language.
- An excellent and complete source of information about programming in R is the following book:
- H. Wickham, M. Çetinkaya-Rundel, G. Grolemund: *R for Data Science*, O'Reilly
- It is openly available at: <https://r4ds.hadley.nz/>
- Many information and examples from this lecture are based on this book.

# Getting started with R

- R (download from <https://cloud.r-project.org>)
- RStudio - IDE (download from <https://posit.co/download/rstudio-desktop/>)
  - RStudio is the IDE that I will use during the lectures, and this is where the screenshots will come from. Obviously, there are other IDEs for R as well, which can be used.

```
1 data()  
2
```

R 43.0 - ~/R

R version 4.3.0 (2023-04-21 ucrt) -- "Already Tomorrow"  
Copyright (C) 2023 The R Foundation for Statistical Computing  
Platform: x86\_64-w64-mingw32/x64 (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.  
You are welcome to redistribute it under certain conditions.  
Type 'license()' or 'licence()' for distribution details.

R is a collaborative project with many contributors.  
Type 'contributors()' for more information and  
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or  
'help.start()' for an HTML browser interface to help.  
Type 'q()' to quit R.

[workspace loaded from ~/.Rdata]

- RStudio IDE has a few important panes:
  - Console (you can directly type commands here)
  - Source (you can write the scripts here)
  - Plots (you will see the visualizations here)
  - Packages (you can see the list of all packages here)
  - Help (where you can read the documentation for functions / data sets)

# Packages

- A package is a collection of reproducible R code, which includes functions, documentation but also data sets.
- There are two very important packages that we are going to use a lot (we will use other packages as well):
  - tidyverse
  - fpp3
- When working with a package, there are two steps:
  - *installing* the package. You only need to do this once, using the following command, (using the name of the package you want to install):

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

- Loading the package, using the *library* function. This has to be done each time you restart RStudio.

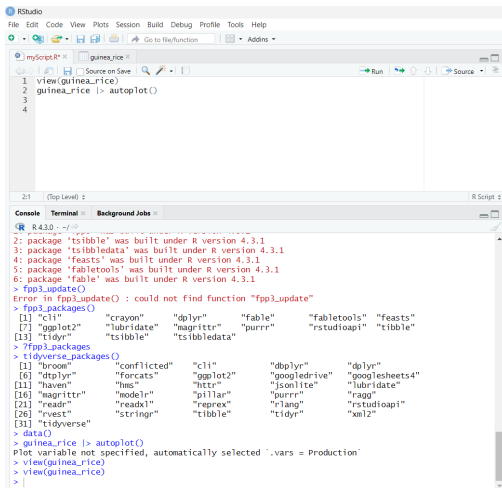
```
library("tidyverse")
```



- When loading a package, other packages can be automatically loaded. And you might get a warning message about conflicts: situations when more packages contain a function with the same name. In this case R tells you which is going to be the *default* function, and which is hidden and needs to be called using the syntax: `package_name::function_name`.

# Console vs. Script

- When working in R, you can write the code either in the Console, or you can create a script: File - New File - R Script
- Scripts can be saved and loaded and re-run later.



The screenshot shows the RStudio environment. The top menu bar includes File, Edit, Code, View, Plots, Session, Build, Debug, Profile, Tools, and Help. Below the menu is a toolbar with icons for file operations and running code. The main editor window displays a script named 'myScript.R' with the following code:

```
1 view(guinea_rice)
2 guinea_rice |> autoplot()
3
4
```

The console window at the bottom shows the output of the script execution. It starts with the R version (4.3.0) and lists several installed packages. It then shows an error message: 'Error in fpp3\_update() : could not find function "fpp3\_update"'. Following the error, it displays the output of the 'fpp3\_packages()' function, which lists various packages. Finally, it shows the output of the 'view(guinea\_rice)' command, which displays a plot of the 'guinea\_rice' data.

```
R 4.3.0 - f
2: package 'tibble' was built under R version 4.3.1
3: package 'tsibbledata' was built under R version 4.3.1
4: package 'feasts' was built under R version 4.3.1
5: package 'fabletools' was built under R version 4.3.1
6: package 'fable' was built under R version 4.3.1
> fpp3_update()
Error in fpp3_update() : could not find function "fpp3_update"
> fpp3_packages()
[1] "crayon"      "dplyr"      "fable"      "fabletools"  "feasts"
[7] "ggplot2"    "lubridate"  "magrittr"   "purrr"      "rstudioapi" "tibble"
[13] "tidyr"      "tsibble"    "tsibbledata"
> ?fpp3_packages
> tidyverse_packages()
[1] "broom"      "conflicted" "cli"        "dbplyr"      "dplyr"
[6] "dtplyr"    "forcats"    "ggplot2"    "googledrive" "googlesheets4"
[11] "haven"     "hms"        "httr"       "jsonlite"    "lubridate"
[16] "magrittr"  "modelr"     "pillar"     "purrr"       "ragg"
[21] "readr"     "readxl"     "reprex"     "rlang"       "rstudioapi"
[26] "rvest"     "stringr"    "tibble"     "tidyr"       "xml2"
[31] "tidyverse"
> data()
> guinea_rice |> autoplot()
Plot variable not specified, automatically selected '.vars = Production'
> view(guinea_rice)
> view(guinea_rice)
>
```

# Some useful functions

- Most packages also contain data sets, not just code. You can see the list of all data sets available using the *data* function (result will be in the upper left panel of R Studio).
- If you want to see the content of a data set, you can use the *view* command.
- ? opens the help for any function or data set.
- For comments we use #

# Data in R

- There are several ways of storing data in R:
  - data frame
  - tibble object
  - tsibble object
- Different packages in R have already a variety of data sets in these formats. There are some transformation functions to go from one representation to another and you can also read data from external files.
- Some useful functions that can be used with any of these representations are:
  - *class* - return type of variable or data set
  - *dim* - return its dimensions (number of observations and variables)
  - *count* - return the number of observations
  - *head* - returns the first few observations (and shows column names)
  - *tail* - returns the last few observations (and shows column names)

# Data frames

- A *data frame* is a representation of simple tabular data.
- You can use \$ to access attributes (by column name)
- Examples of data sets stored as data frame are: *cars*, *iris*

```
class(cars) # will display "data.frame"  
dim(cars) # will display 50 2  
cars$speed # all the values for the speed attribute  
cars[1, 2] # second attribute of the first observation (indexing goes from 1)  
cars[1, ] # the first observation  
cars[, 2] # 2nd attribute for all observations (as a sequence of numeric values)
```

# Reading from csv

- You can read the content of a csv file as a data frame, using the *read.csv* function.

```
df <- read.csv("Brent Spot Price.csv")  
dim(df)  
class(df)  
df$X
```

- The above csv file actually contains a time series, it contains the price of oil for different months. The first column contains values like "1990 01", "1990 02", ..., "2022 12" which are treated as simple strings here.
- So, even though the content is a time series, we are not storing it as a time series in this case (functions specific for time series will not work on it).

- A *tibble* is considered a neater format than a data frame, it contains the same information, but the manipulation and internal representation of tibbles is different.
- Example of a data set stored as tibble is *starwars*.

```
class(starwars) # "tbl_df"      "tbl"        "data.frame"
dim(starwars)  # 87 14
head(starwars) # displays types of attributes as well
starwars$name  # all values for the name attribute
starwars[, 4]  # all values for the fourth attribute as a tibble (87 x 1)
starwars[1, ]  # first observation
starwars[1, 4] # fourth attribute of the first observation

#get the name of the first character
starwars[1, ]$name
starwars[, 1][1, 1]
starwars[, 1][1, ]
```

# Data frame - tibble transformations

- You can read the content of a csv file as a tibble using the *read\_csv* function.
- It will return an *spec\_tbl\_df* object, which is a regular tibble, but it has also column specifications. You can use the *spec* function to see them.
- You can transform a data frame into a tibble with the *as\_tibble* function (and transform a tibble to a data frame with *as.data.frame*).
- **Obs:** package *readxl* has a *read\_excel* function which can read directly excel files.



```

tb <- read_csv("Brent Spot Price.csv")
class(tb) #spec_tbl_df "tbl_df"      "tbl"        "data.frame"

tb_df <- as.data.frame(tb)
class(tb_df)

df_tb <- as_tibble(df) #from a few slides ago, the result of the read_csv
                        function
df_tb
class(df_tb)

?read_csv
library("readxl")
?read_excel
tb2 <- read_excel("wineE.xlsx")
tb2
class(tb2)

```

# Tsibble objects

- It is a Time series tibble.
- Compared to other time series objects (for example `ts`), it allows heterogeneous data.
- It contains some measurements and information about the time when the measurements were recorded (the index).
- An example of a tsibble from R is the *olympic\_running* data set, which contains data about the fastest running times at the Olympic games.

```

olympic_running
head(olympic_running) # will print the dimensions, [4Y] to show that it has 4
                        year frequency, column names and types and the first few instances
dim(olympic_running) #will print 312 4
olympic_running$Year # all the values for the year attribute
olympic_running[, 2] # values of the second attribute (as a 312 x 1 tibble)
class(olympic_running) # "tbl_ts"          "tbl_df"          "tbl"          "data.frame"

```

- Every tsibble has to contain an index, this is the timestamp of the observations. For the *olympic\_running* example, it is obviously the year. However, if we look at all the values from the year attribute, we can see that it is not unique. The reason for this is that the *olympic\_running* tsibble actually contains several time series. For every combination of *length* and *sex* we have a separate time series. Thus, *length* and *sex* are the *keys* of the tsibble, and we can see that we actually have 14 distinct time series.

# Creating from data

- While most of the time we read data from csv, there are also functions to create them from existing values.
- function *data.frame* can be used to create data frames and *tibble* can be used to create tibbles. You just enumerate the columns for both functions.

```
x <- data.frame(  
  a = 2:10,  
  b = c('a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i')  
)  
x  
class(x)  
  
y <- tibble( a = 2:10,  
             b = 12:20,  
             c = c("a", "b", "c", "d", "e", "f", "g", "h", "i"))  
y  
class(y)
```

- **Obs.:** The *c* function combines the given values in a vector .
- **Obs2.:** The *:* (colon) operator will return a sequence (3:7 will return the sequence 3 4 5 6 7)

- You can create a tsibble using the *tsibble* function, where you need to specify all the attributes and their values and then specify which attribute is the *index*, the one showing the time frequency. If your data has keys, those can be specified as well.

```
y <- tsibble(  
  Year = 2015:2019,  
  Observation = c(11, 52, 31, 76, 12),  
  index = Year)  
y # will print the size (5x2), will print that we have yearly data, [1Y], and  
   the data  
  
w <- tsibble(  
  Year = c(2011, 2012, 2013, 2014, 2011, 2012, 2013, 2014),  
  Specializations = c("CS", "CS", "CS", "CS", "DS", "DS", "DS", "DS"),  
  NrStudents = c(20, 25, 30, 35, 23, 32, 19, 34),  
  index = Year,  
  key = Specializations  
)  
w
```

- For other frequencies, specific time class functions might be used to build the index:

Frequency	Function from string
Annual	start:end
Quarterly	yearquarter()
Monthly	yearmonth()
Weekly	yearweek()
Daily	as_date(), ymd()
Sub-daily	as_datetime(), ymd_hms()

- The functions from column 2 receive as parameter a string, which represents a *date*, for example "1994 Q3" or "2014 August" and return an object representing that date.
- Most of the functions have versions which receive as parameter integer values representing components of the date and create a date object (see examples below).
- They can work on one individual value or an entire column, if needed.

- Examples of transformations using *yearquarter*

```
q1 <- yearquarter("2010 Q1")
q2 <- yearquarter("2010 Quarter 2")
q3 <- yearquarter("2010-04-01") #will be 2010 Q2
q4 <- yearquarter("2010-04-01", fiscal_start = 7) # fiscal_start represents the
    month when fiscal year starts (where you count quarters from), so this will
    be 2010 Q4
q5 <- make_yearquarter(year = 2010, quarter = 4) #create year quarter from
    integer numbers
q6 <- make_yearquarter(year = rep(2010:2019, each = 4), quarter=rep(1:4, 10))
```

- The function *rep* replicates the values of a vector, so *rep(1:4, 10)* will replicate the vector [1,2,3,4] 10 times (result being [1,2,3,4,1,2,3,4, ...]), while parameter *each* determines how many times to repeat a value consecutively (so *rep(2010:2019, each=4)* will be [2010, 2010, 2010, 2010, 2011, 2011, ...]).
- *q6* will be a vector of 40 yearquarter objects, starting from 2010 Q1 to 2019 Q4.

## ● Examples of transformations using *yearmonth*

```
m1 <- yearmonth("2010 April")
m2 <- yearmonth("2010 07")
m3 <- yearmonth("2010-08-29")
m4 <- yearmonth("October 2010")
m5 <- yearmonth("22-05-2023") #error, "22-05-2023" cannot be expressed as Date
    type
m5 <- yearmonth("22-05-2023", format="%d-%m-%y") #works
m6 <- make_yearmonth(year = 2020, month = 11)
m7 <- make_yearmonth(year = rep(2020:2023, each=12), month = rep(1:12, 4)) # a
    vector of 48 yearmonth objects
m8 <- yearmonth("2020 M06")
```



- Examples of transformations using *yearweek*

```
w1 <- yearweek("2020 Week4")
w2 <- yearweek("2020 W25")
w3 <- yearweek("2023-10-12")
w4 <- yearweek("2023-10-12", week_start = 7)
w5 <- yearweek("2023 Wk40")
w6 <- yearweek("2023-02") #02 is for Febr. — will consider the first of Febr.
w7 <- make_yearweek(year = 2022, week = 5)
w8 <- make_yearweek(year = 2022, week = 60) #result will be 2023 Week 8
w9 <- make_yearweek(year = 2020, week = 53) # will be 2020 Week 53
w10 <- make_yearweek(year = 2021, week = 53) # will be 2022 Week 1
is_53weeks(2021) #FALSE
is_53weeks(2020) #TRUE
```

- By default a week starts on Monday, but with the *week\_start* parameter we can change that. In w4 the week starts on Sunday.
- Mathematically in a year there are 52.14 or 52.29 (leap year) weeks. Since we need an integer number of weeks, ISO year format has exactly 52 or 53 (once in approx 5-6 year) weeks.

- Examples of transformations using *as\_date* and *ymd*

```
d1 <- as_date("2023-09-30")
d2 <- as_date("2023 09 25")
d3 <- ymd("2020-06-15")
d4 <- ymd(20200605)
d5 <- ymd("Date is 2020 12 22")
s1 <- seq(as_date("2020-01-01"), as_date("2020-03-20"), by = "1 day") # a vector
  of 73 consecutive days
weeks <- seq(as_date("2020-01-01"), as_date("2020-03-20"), by = "1 week") |>
  yearweek() #12 consecutive weeks
months <- seq(as_date("2020-01-01"), as_date("2020-06-30"), by = "1 month") |>
  yearmonth() # 6 consecutive months
```

- *ymd* has variations, *ydm*, *myd*, *mdy*, *dym* and *dmy*.

- Examples of transformations using *as\_datetime* and *ymd\_hms*

```
dt1 <- ymd_hms("2023-09-30 01:00:00")  
dt2 <- ymd_hm("2023-09-30 02:50")  
dt3 <- as_datetime("2013-05-25 14:10:00.00")
```

- As can be seen from the above example, *ymd\_hms* has versions for cases where seconds or minutes and seconds are not needed: *ymd\_h* and *ymd\_hm*

- In many cases, we want to perform multiple operations, each one on the result of the previous one. Writing this as one instruction leads to nesting, which makes understanding the code more complicated.
- A better solution is the *pipe* operator, `|>` which simply means *"take the result of what is on the left and use it as first parameter in what is on the right"*.
- When using pipe we can also use right arrow assignment, specifying that we want the result to be stored in the given variable.

- Let's see how we can transform a data frame into a tsibble, using piping.
- **Obs.** Function *mutate* changes the value of the attributes in a column.

```
read.csv("Brent Spot Price.csv") |> # it is a data frame
  mutate(X = yearmonth(X)) |> #change the X column to a monthly time object
  as_tsibble(index=X) -> exTsibble #create tsibble setting the new X as index
  and save the result in exTsibble
exTsibble # will show that we have a 396x2 tsibble with monthly data
```

# Useful functions for working with tabular data I

- Tsibbles allow multiple attributes to be stored for an observation and also a tsibble can contain several time series. These time series are identified by a variable (or combination of variables) called keys.
- As mentioned, the *olympic\_running* tsibble contains the fastest running times from the olympic games for several distances, both for men and women, each of them being a separate time series, for example: running times for 100 meters men
- If we print such a tsibble, it will contain the keys for the data (for *olympic\_running* Length and Sex) and how many time series are in the data (14 in our case)
- Let's see some useful operations for working with such tsibbles:

# Useful functions for working with tabular data II

- `mutate()` - change the values for an attribute

```
read.csv("Brent Spot Price.csv") |> # it is a data frame
mutate(X = yearmonth(X)) |> #change the X column to a monthly
time object
mutate(Price = Brent.crude.oil.spot.price..Monthly..dollars.per.
barrel.) |> #technically we create a new attribute with same
data but different name
as_tibble(index=X) -> exTibble #create tibble setting the new
X as index and save the result in exTibble
```

- You can create new attributes by combining the value of existing ones

```
olympic_running |>
mutate(AverageTime = (Time / Length)) -> olympic_running_average
```

# Useful functions for working with tabular data III

- `select()` - select only some columns (useful when there are many attributes and we are interested only in some of them)

```
read.csv("Brent Spot Price.csv") |>
  mutate(X = yearmonth(X)) |>
  mutate(Price = Brent.crude.oil.spot.price..Monthly..dollars.per.
    barrel. ) |>
  select(X, Price) |>
  as_tibble(index = X) -> exTibble
```

- Can also be used to reorder the columns
- If instead of selecting some columns, we want to rather eliminate something, we can use the minus sign and the name of the column.



# Useful functions for working with tabular data IV

```
#let's create a tsibble with some artificial columns
exTsibble <- read.csv("Brent Spot Price.csv") |>
  mutate(Date = yearmonth(X)) |>
  mutate(Price = Brent.crude.oil.spot.price..Monthly..dollars.per.
    barrel. ) |>
  mutate(Price2 = Price) |>
  as_tsibble(index = Date)

exTsibble |>
  select(Price, Date)

exTsibble |>
  select(-X, -Brent.crude.oil.spot.price..Monthly..dollars.per.
    barrel.)
```

- You cannot eliminate (not select) the columns which are the index and the key of the tsibble (they will be added at the end of the attributes anyway)

# Useful functions for working with tabular data V

- `distinct()` - will return the distinct values for an attribute or combination of attributes, in the form of a tibble

```
olympic_running |> distinct(Sex)

olympic_running |> distinct(Sex, Length)

olympic_running |> distinct(Year) |> print(n = 31) #instead of the
standard 10, print 31 observations
```

# Useful functions for working with tabular data VI

- `filter()` - extract a part of the data based on the value of one or more attributes
- An important possible condition is *is.na* which checks if a value is NA (Not Available). We can use this to filter out the NA values which might create problems later.

```
olympic_running |> filter(Sex == "women")  
olympic_running |> filter(Year > 1960)  
olympic_running |> filter(Year > 1960, Sex == "women")  
olympic_running |> mutate(Average = Length / Time) |> filter(Average < 15)  
view(olympic_running |> filter(is.na(Time)==FALSE))
```

# Useful functions for working with tabular data VII

- `slice()` - returns a part of the observations. Similar to `filter`, except that you select the elements based on index, not based on a condition.

```
olympic_running |> slice(1:20)
```

# Useful functions for working with tabular data VIII

- `unite()` - to combine two or more columns into a single column. You need to specify the data frame, the name of the new column, what columns to unite and what separator to use.

```
olympic_running |>
  unite(Name, Length, Sex, sep = "-") -> olympic_name

olympic_name

olympic_running |>
  unite(Name, Length, Sex, sep = "-", remove = FALSE) -> olympic_
  name

olympic_name

olympic_name |>
  as_tibble(index = Year, key = Name)
```

# Other important functions

- *summary* returns a 5-number summary of the data (minimum, maximum, 1st Quartile, median, mean, 3rd Quartile)
  - You can call it either for one attribute of the data, or for an entire tibble/tsibble
- *mean*, *sum*, *sd* (standard deviation)
  - These work only on one attribute

```
wines <- read_csv("wine.csv") #read as a tibble
summary(wines)
summary(wines$Rose)
sum(wines) #error
sum(wines$Rose)

#how many Rose wines were sold in 1985
wines |>
  filter(Year == 1985) |>
  select(Rose) |>
  sum()
```

# Data visualization - simple visualizations I

- For visualization the *ggplot2* system is used. Besides being able to create very complex visualizations, *ggplot2* has a nice, simple function called *autoplot* which simply chooses the type of plot based on the data you pass to it.
- If we provide a time series to *autoplot*, it will automatically create a line plot.

```
men100 <- olympic_running |> filter(Sex == "men", Length == 100)
autoplot(men100)
```

# Data visualization - simple visualizations II

- When there are several numerical attributes, autoplot will automatically select the one to plot, which might not be the one you want. You can specify which attribute to plot by passing its name as a parameter to *autoplot*.

```
olympic_running_average |>  
  filter(Sex == "men", Length == 100) |>  
  autoplot(AverageTime)
```



# Data visualization - simple visualizations III

- autoplot will add some axis labels as well, but we can customize what to be added there, using the *labs()* function.
  - For *labs* you can specify text for the following:
    - x
    - y
    - title
    - subtitle
    - caption
  - The + sign is used to add layers to a plot. Labels are considered to be layers. If you divide your code on several lines, make sure that the + is at the end of the line not at the beginning!

```
autoplot(men100) +  
  labs(x = "Time", y = "Running time", title = "Running times at the olympic  
    games", subtitle = "100 meters men", caption = "Plot of running times for  
    the 100 meters men at the olympic games from 1896 to 2016")
```

# Data visualization - simple visualizations IV

- *autoplot* does not support data of type data frame or tibble. If we want to plot the *wine* data, for example, we need to transform it to a tsibble. Transformation can be done with *as\_tsibble*, but we need to specify the index for the time series. We have monthly data, and we have a column for Year (numeric) and one for Month (character). For *yearmonth* function we need character representation, so we need to *unite* them.

```
autoplot(wines) # error.  
winesTS <- wines |>  
  unite(Date, Year, Month, sep = " ") |>  
  mutate(Date = yearmonth(Date)) |>  
  as_tsibble(index = Date)  
autoplot(winesTS) #will plot Fortified as time series  
autoplot(winesTS, Red) # will plot Red as time series  
winesTS |>  
  slice(1:24) |>  
  autoplot(Total) #zoom in on the first two years
```

# Data visualization - simple visualizations V

- If we want to plot several time series with autoplot, we can use *autolayer*. However, in this case, piping is not going to work, autolayer needs to receive as first param the tsibble to work with (and param 2 the attribute).
- You can add colours, to display different time series in different colours.

```
autoplot(winesTS, Total) + autolayer(winesTS, Red)
autoplot(winesTS, Total, color = "Green") +
  autolayer(winesTS, Red, colour="Red") +
  autolayer(winesTS, Fortified, colour="Blue")
```

# Data visualization - simple visualizations VI

- Today, we have worked with two tsibbles containing several times series: *olympic\_running* and *wines*. What happens if we autoplot them?

```
autoplot(olympic_running)  
autoplot(winesTS)
```

# Wide and long data formats I

- While both tibble contain several time series, they are structured differently:
- For *wines*
  - we have a row for each year, and we have columns for different types of wines (attributes).
  - each row describes a unique year.
  - this is called a **wide** data format (many columns, less rows).
- For *olympic\_running*:
  - the same year can appear on multiple lines, the exact time series is determined by the keys (Sex and Length).
  - this is called a **long** data format (less columns, more rows).

# Wide and long data formats II

- Advantages of the *long* format
  - In case of missing data no space is wasted (missing data is just rows that do not exist)
  - If new attributes are added, the structure of the table will not be changed, only new rows need to be added with the new attributes.
  - autoplot handles them nicely
- Advantages of the *wide* format
  - Our brain processes it easier
    - ex. comparing the values of different attributes over all observations

# Wide and long data formats III

- In R we can transform one data format to another, using the *pivot\_longer* and *pivot\_wider* functions.
  - *pivot\_longer* - makes the data longer, by transforming an attribute into rows. Besides the data, you need to specify the name of the column which will contain the attribute names (the *names\_to* parameter) and the name of the column which will contain the values (the *values\_to* parameter).
  - For transforming several attributes to rows at once, we combine the column names to be transformed.

```
winesTS |> pivot_longer(Red, names_to = "Type", values_to = "QuantitySold")  
      -> winesLonger  
  
winesTS |> pivot_longer(c(Fortified, Drywhite, Sweetwhite, Rose, Red,  
  Sparkling, Total), names_to = "Type", values_to = "QuantitySold")  
      ->  
winesLong
```

# Wide and long data formats IV

- For transforming into the "wide" format, we need to specify the values of what attribute should be transformed into columns (*names\_from* parameter) and what values to put in those columns (*values\_from* parameter).

```
olympic_running |> pivot_wider(names_from= Sex, values_from = Time) ->  
  olympic_running_wider  
  
olympic_running |> pivot_wider(names_from= c(Sex), values_from = Time) |>  
  pivot_wider(names_from = c(Length), values_from= c(men, women)) ->  
  olympic_running_wider  
  
autoplot(olympic_running_wider)
```



- For visualization the *ggplot2* system is used, starting from the function *ggplot()* which defined a plot object, to which we can add **layers**.
- First parameter of *ggplot* is the data set which is going to be plot.
- Then, you need to provide a *mapping* telling *ggplot* which variables to use and what *aesthetics* (visual properties) to link them to (you will see an example immediately).
- You also need a *geom*, a geometrical object that is used to represent data (you can think of it as the type of the plot: bar chart uses *geom\_bar()*, line chart uses *geom\_line()*, boxplot uses *geom\_boxplot()*, scatterplot uses *geom\_point()*, etc.)

# Penguins example

- Consider the *palmer penguins* data set, a classical example used for data visualizations.
- It contains body measurements for 344 penguins, for each of them containing the following information:
  - species (Adelie, Gentoo, Chinstrap)
  - island where the penguin lives (Torgersen, Biscoe, Dream)
  - length and depth of the bill
  - length of the flipper
  - body mass
  - sex
  - year when the measurement was made (2007, 2008, 2009)
- You can get the data set either by installing the *palmerpenguins* package in R, or from a csv file.

```
penguins <- read_csv("penguins.csv")
```

- The goal is to get a visual representation of the relationship between the body mass and the flipper length of the penguins and see if this relationship changes depending on the species of the penguins.
- Let's start plotting, by creating an empty plot to which we will add layers.
- The first parameter to ggplot is the data that we use.

```
ggplot(data = penguins)
```

- The second important parameter of *ggplot* is the *mapping* which is done through *aesthetics*.
- An *aesthetic* is "something you can see", so the goal of the mapping is to connect variables to visual cues.
- One of the most important visual cues are the axes, so the first thing we need to specify is what data to plot on each axes.
- We want the attributes *flipper\_length\_mm* and *body\_mass\_g* to be plot on the *x* and *y* axes.

```
ggplot(data = penguins ,  
       mapping = aes(x = flipper_length_mm, y = body_mass_g))
```

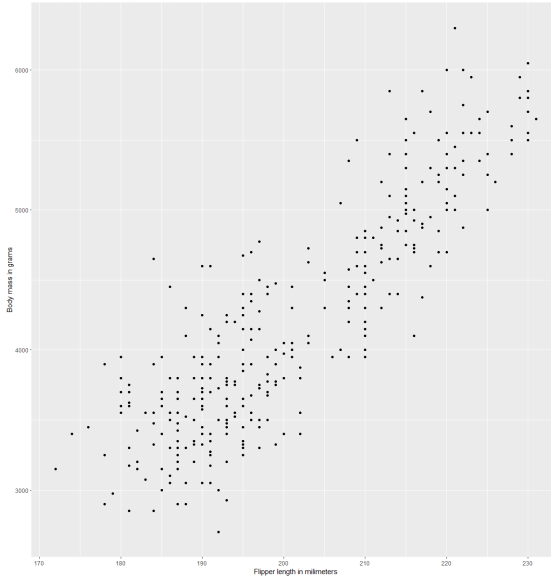
- We can see the labels on the axes, but no data, since we did not tell *ggplot* yet how to visualize the penguins.

- Since we want a scatterplot, we will use *geom\_point()*.

```
ggplot(data = penguins, mapping = aes(x = flipper_length_mm, y = body_mass_g)) +  
  geom_point() +  
  labs(title = "Plot of penguin flipper length vs body mass", x = "Flipper  
length in millimeters", y = "Body mass in grams")
```

- We will get a warning about removing two instances with NA values, but the rest of the instances were.

Plot of penguin flipper length vs body mass



- From the plot it seems like there is a positive, quite linear relation between flipper length and body mass.
- Note: we can check the correlation between the two attributes to see how strong the correlation is, using the `cor` function.
- However, the correlation will be NA, since we have instances with NA values. So, first we need to remove them, using the `filter` function with the `is.na` function as condition.

```
penguins |> filter(is.na(flipper_length_mm) == FALSE & is.na(body_mass_g) == FALSE) -> penguinsNoNA  
  
cor(penguinsNoNA$flipper_length_mm, penguinsNoNA$body_mass_g)
```

- We have a correlation of 0.87 which is a strong, positive correlation. Let's see if this relation changes depending on the species of the penguin. For this, we will add the *Species* attribute to our aesthetics and assign the color of the dots to them.
- When a categorical variable is mapped to an aesthetic, for each distinct value of that variable a unique aesthetic value will be mapped and a legend is also added to the plot.

```
ggplot(data = penguins,
       mapping = aes(x = flipper_length_mm, y = body_mass_g, color = species)) +
  geom_point() +
  labs(title = "Plot of penguin flipper length vs body mass", x = "Flipper
length in millimeters", y = "Body mass in grams")
```



Plot of penguin flipper length vs body mass



- If we want, we can recompute our correlations separately for the three species as well:

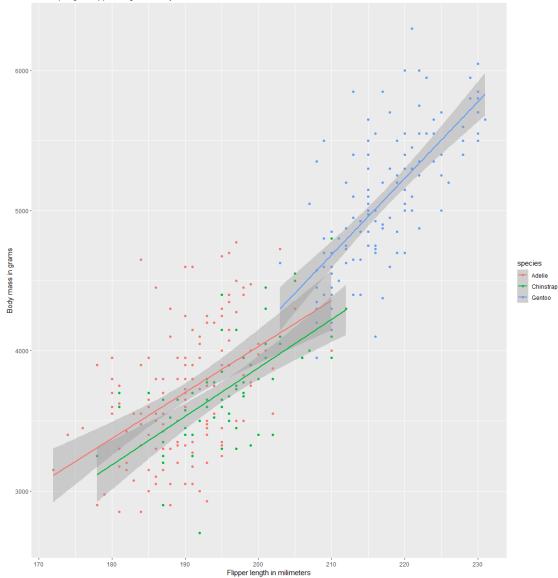
```
penguinsNoNA |> filter(species == "Adelie") -> penguinsAdelie
penguinsNoNA |> filter(species == "Chinstrap") -> penguinsChinstrap
penguinsNoNA |> filter(species == "Gentoo") -> penguinsGentoo

cor(penguinsAdelie$flipper_length_mm, penguinsAdelie$body_mass_g)
cor(penguinsChinstrap$flipper_length_mm, penguinsChinstrap$body_mass_g)
cor(penguinsGentoo$flipper_length_mm, penguinsGentoo$body_mass_g)
```

- As the next step, let's add a smooth curve displaying the relationship between body mass and flipper length. For this, we need to add a new layer to our plot (besides the one created by `geom_point()`), of type `geom_smooth()`. We need to specify what method to be used to create the smooth line. In this example we use a *linear model*, `lm`.

```
ggplot(data = penguins,
       mapping = aes(x = flipper_length_mm, y = body_mass_g, color = species)) +
  geom_point() +
  geom_smooth(method = "lm")
```

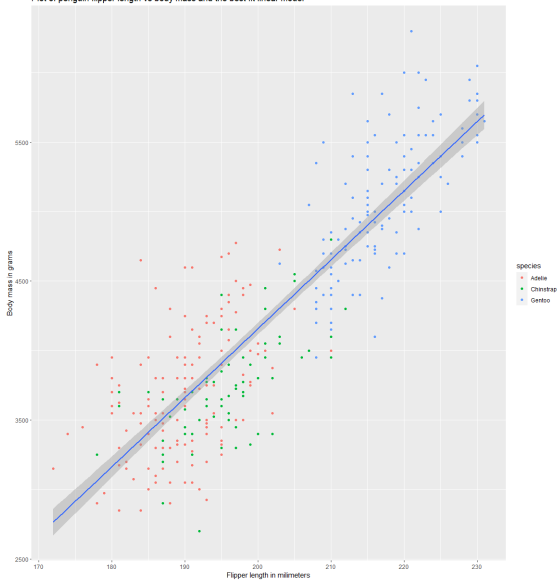
Plot of penguin flipper length vs body mass and the best fit linear model



- We have three smooth lines, one for each species, but we wanted only one. The reason for having three lines is that the aesthetic mapping that we have specified works at *global* level, all layers inherit it. If we want some aesthetics to be valid on the *local* level only, we can add that aesthetics in the *geom* function.

```
ggplot(data = penguins,
       mapping = aes(x = flipper_length_mm, y = body_mass_g)) +
  geom_point(mapping = aes(color = species)) +
  geom_smooth(method = "lm")
```

Plot of penguin flipper length vs body mass and the best fit linear model



- The shaded region around the line is the confidence interval for the line. We can set the level of the confidence interval to use with the *level* parameter, the default is 0.95.
- We can specify to not draw a confidence interval, setting the parameter *se* to false.
- **Obs.** Technically, *se* is not a confidence interval, it is a *standard error*.

```
ggplot(data = penguins, mapping = aes(x = flipper_length_mm, y = body_mass_g)) +
  geom_point(mapping = aes(color = species)) +
  geom_smooth(method = "lm", se = FALSE) +
  labs(title = "Plot of penguin flipper length vs body mass and the best fit
           linear model", x = "Flipper length in milimeters", y = "Body mass in grams"
        )

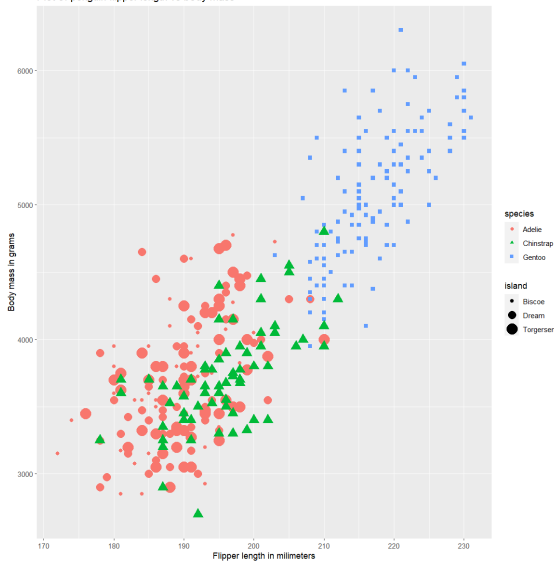
ggplot(data = penguins, mapping = aes(x = flipper_length_mm, y = body_mass_g)) +
  geom_point(mapping = aes(color = species)) +
  geom_smooth(method = "lm", level = 0.99) +
  labs(title = "Plot of penguin flipper length vs body mass and the best fit
           linear model", x = "Flipper length in milimeters", y = "Body mass in grams"
        )
```

- Some other aesthetics that can be used:
  - shape - only for discrete attributes
  - size - not advised for discrete attributes
  - alpha (how transparent it should be) - not advised for discrete attributes
- While not every combination works, you can map the same attribute to more than one aesthetic.

```
ggplot(data = penguins, mapping = aes(x = flipper_length_mm, y = body_mass_g,  
  color = species, size = bill_length_mm, alpha = bill_length_mm)) +  
  geom_point() +  
  labs(title = "Plot of penguin flipper length vs body mass", x = "Flipper  
    length in millimeters", y = "Body mass in grams")  
  
ggplot(data = penguins, mapping = aes(x = flipper_length_mm, y = body_mass_g,  
  color = species, size = sex)) +  
  geom_point() +  
  labs(title = "Plot of penguin flipper length vs body mass", x = "Flipper  
    length in millimeters", y = "Body mass in grams")
```



Plot of penguin flipper length vs body mass



- We can assign constant values to an aesthetic as well. In our initial scatter plot, when the color was not mapped to the species of the penguins, all points were black. We can change this color to another one if we want to. In order to use constant values, the assignment needs to be in the `geom_point` function, but not in the aesthetics call.

```
ggplot(data = penguins, mapping = aes(x = flipper_length_mm, y = body_mass_g))  
+  
geom_point(shape = "square", color = "tan") +  
labs(title = "Plot of penguin flipper length vs body mass", x = "Flipper  
length in millimeters", y = "Body mass in grams")
```

- The above plot will display for every penguin a square on the plot having a tan color.
- **Obs.** There are 657 color defined with a name in R. You can list them with the *colors* function.

- When we map an attribute to an aesthetic (for example, map species to color), R will automatically choose the used aesthetic values (the exact colors). In our case, it chose red, blue and green for species.
- We can change these colors, by adding a *scale* layer to our plot. For our example, it should be a *scale\_color\_manual* (we manually set the colors to be used). You need to provide at least as many colors as the possible values for the attributes.

```
ggplot(data = penguins, mapping = aes(x = flipper_length_mm, y = body_mass_g,
  color = species)) +
  geom_point(shape = "square") +
  labs(title = "Plot of penguin flipper length vs body mass", x = "Flipper
  length in millimeters", y = "Body mass in grams") +
  scale_color_manual(values = c("magenta", "salmon", "tan", "yellow"))
```

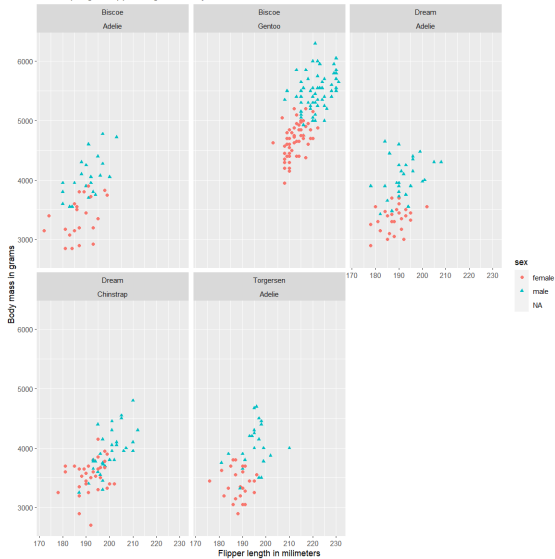
- Let's assign the shape and color to the species and the island to the size. At this point we have 4 information on the plot and we might consider it hard to understand.
- If we have a categorical variable, we can split the plot into subplots, each containing the visualization of the data points having one specific value for the categorical variable. This can be done with *facet\_wrap()*, which takes as parameter the name of the variable to use for splitting.

```
ggplot(data = penguins, mapping = aes(x = flipper_length_mm, y = body_mass_g,
  color = species, shape = species)) +
  geom_point() +
  facet_wrap(vars(island)) +
  labs(title = "Plot of penguin flipper length vs body mass", x = "Flipper
  length in millimeters", y = "Body mass in grams")
```

- Facet wrap arranges automatically the facets in rows and columns, but we can change this with the *nrow* and *ncol* parameters.
- We can also split in facets by several attributes.

```
ggplot(data = penguins, mapping = aes(x = flipper_length_mm, y = body_mass_g,  
  color = sex, shape = sex)) +  
  geom_point() +  
  facet_wrap(vars(island, species), nrow = 3, ncol = 3) +  
  labs(title = "Plot of penguin flipper length vs body mass", x = "Flipper  
  length in millimeters", y = "Body mass in grams")
```

Plot of penguin flipper length vs body mass



# Other useful general chart types

- To see the distribution of categorical variables we can use *bar chart*, with the command `geom_bar()`.

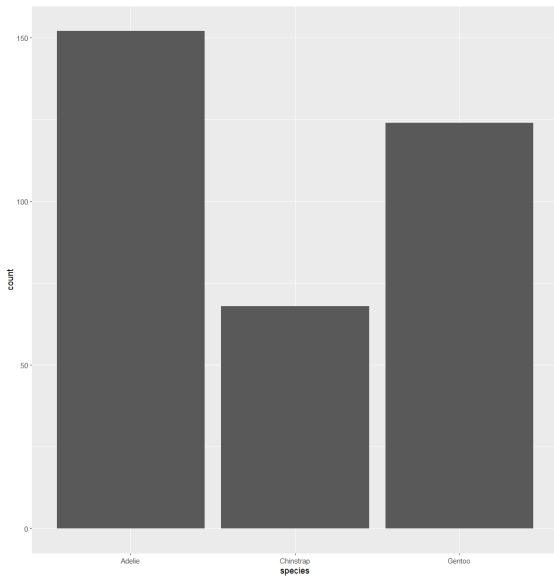
```
ggplot(data = penguins, mapping = aes(x = species)) +  
  geom_bar()
```

- **Obs.** Note how we only specify the data for the *x* axis. On the *y* axis, by default we have the counts for the values of the variable.
- One interesting aesthetic of bar charts, is *fill*, which can be used to build stacked bar charts, if you assign to it a categorical attribute.

```
ggplot(data = penguins, mapping = aes(x = species)) +  
  geom_bar(color = "red")
```

```
ggplot(data = penguins, mapping = aes(x = species)) +  
  geom_bar(color = "red", fill = "red")
```

```
ggplot(data = penguins, mapping = aes(x = species)) +  
  geom_bar(mapping = aes(fill = island))
```

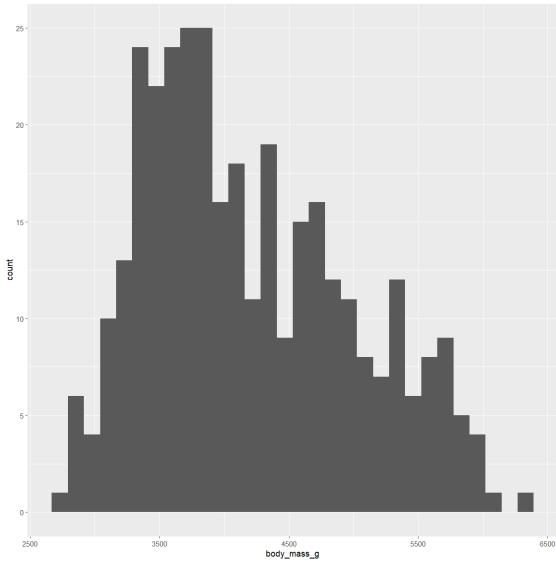




- For seeing the distribution of numerical variables, we can use a *histogram*, with the *geom\_histogram()* command.
- This command needs a parameter specifying the width of a bin from the histogram. It is a good idea to experiment with several different values.

```
ggplot(data = penguins, mapping = aes(x = body_mass_g)) +  
  geom_histogram()
```

Histogram of body mass considering bins with a width of 30



# Saving the plots

- The *ggsave* function will save the most recently created plot on the computer.

```
ggplot(data = penguins, mapping = aes(x = body_mass_g)) +  
  geom_histogram(binwidth = 200)  
ggsave(filename = "penguin-plot.png")
```

- There are several parameters that can be set (including where to save, background, dpi, etc.), you can see them in the documentation (opened by *?ggsave*).
- By default, *ggsave* will save the last plot. We can save other plots in the following way:
  - Save the *ggplot* object in a variable
  - Specify the name of the variable as the *plot* parameter for *ggsave*

```
myplot <- ggplot(data = penguins, mapping = aes(x = species)) +  
  geom_bar()  
#do other stuff here  
ggsave(filename = "penguin-plot2.png", plot = myplot)
```

- There is a really nice and detailed cheatsheet about ggplot2 (all types of plots, their aesthetics, etc.) at the following link:  
<https://rstudio.github.io/cheatsheets/data-visualization.pdf>
- They have a nice cheatsheet about data transformations and data cleaning as well:  
<https://rstudio.github.io/cheatsheets/data-transformation.pdf>,  
<https://rstudio.github.io/cheatsheets/tidyr.pdf>