Data wrangling

Miriam Lerma

May 2023

Index

Today

Your profile

- You have R and Rstudio installed
- You can navigate inside Rstudio

Goals of today

- Difference between R script and Rmd
- Load data
- Basic operations
- Manipulate data
- Export clean data

Pauses and questions

- Exercises and 10 minute pauses for catching up
- You can stop me to ask questions or use this link

References

- R for Data Science
  R4DS
- Data Carpentries
  Carpentries
- R cookbook
  R cookbook
- From Zero to Shero by RLadies
  Zero to Hero
- Images from
  Unsplash
  Allison horst

R markdown

Parts of the kitchen

1. RMarkdown

Rmarkdown is very convenient because it let us export what we do in R to html or word documents.

We can even make slides...

...like this one.

There are **plenty** of options on Rmd. I will just point out a few.

1.2. Rmd

Markdown is plain text...

...just as we write in a note block.

However, the advantages of using Rmd is that you can include a lot of text and thus you can write your **thesis, papers, webpage, books and presentations** without leaving RStudio.

Moreover, you can include:

- code and results that are automatically generated.

1.2. Rmd

What Rmd does, is that it "translates" what has been written using **PanDoc**.

1.2. Rmd

We can generate an **output**, that can be read even if you dont have R install.

Just like this presentation.

Also, you can get your results without showing the code and without having to copy and paste the results in other program like word.

- Download Rmd file here.

See the example:

List of ingredients

```
ingredients<-c('tomatoes','onions','pepper','salt','oil')
length(ingredients)
```

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

1.2. R vs Rmd

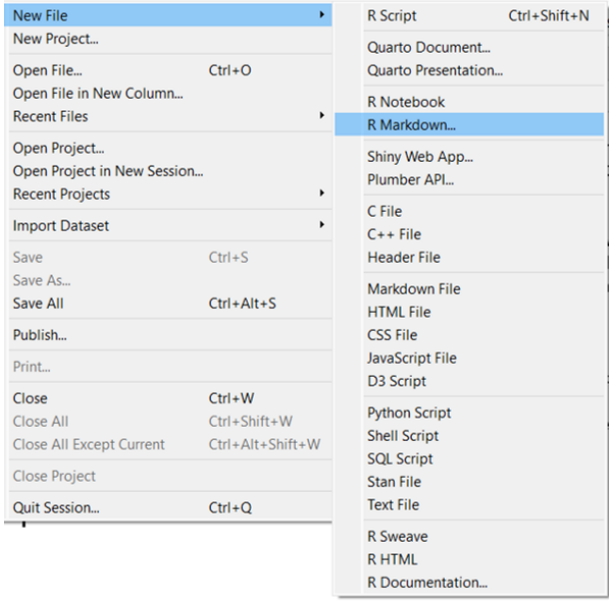
Considerations:
Rmd behaves differently than R

- Rmarkdown works better in a clean environment.
- All the variables need to be inside your file.
- This actually assures that your workflow is **reproducible**.

1.3. Start an Rmd

To start a new file

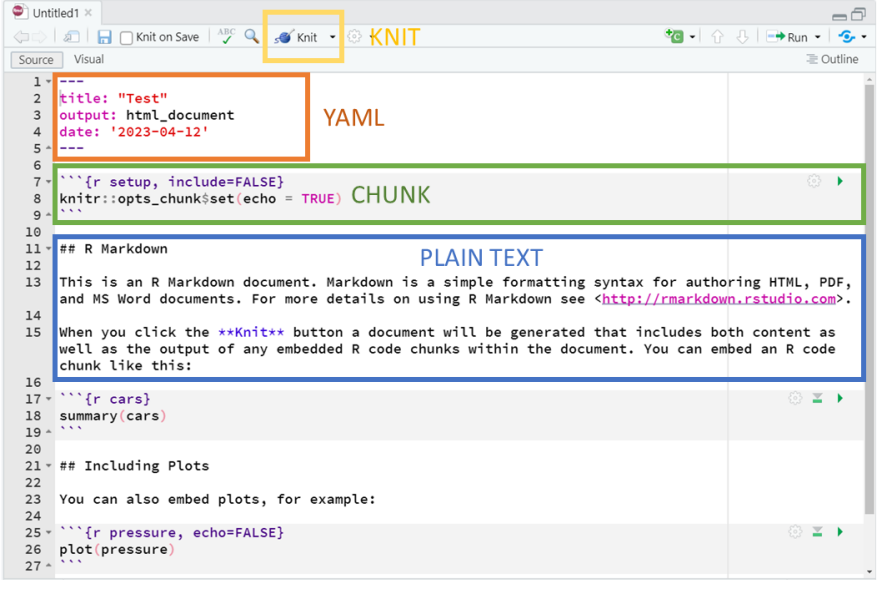
File>NewFile>RMarkdown



1.4. Rmd parts

Rmd has four main parts:

- yaml (including the output)
- chunks
- plain text
- knit

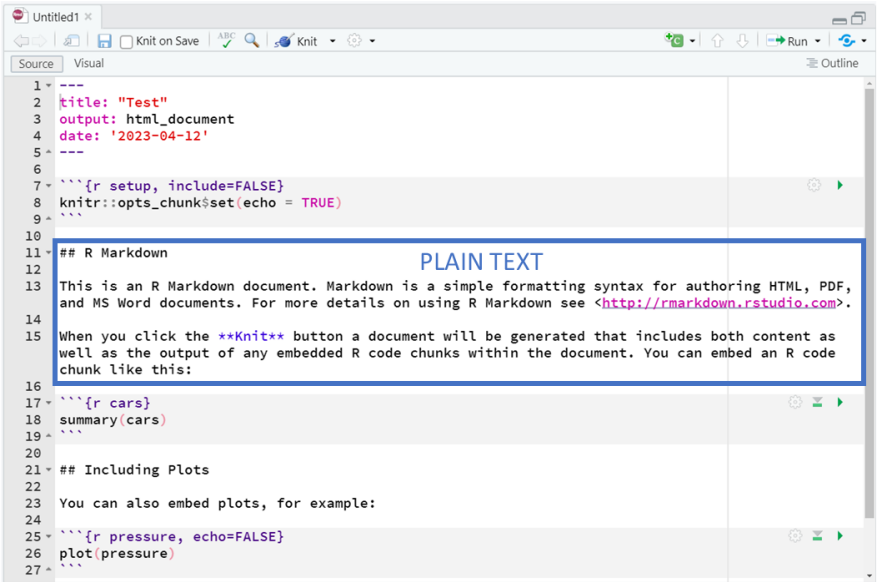


1.5. Rmd text

You can write plain text in the white area

You can use:

- **bold** using two astheriscs.
- *italics* using one astherics before and one after the word.

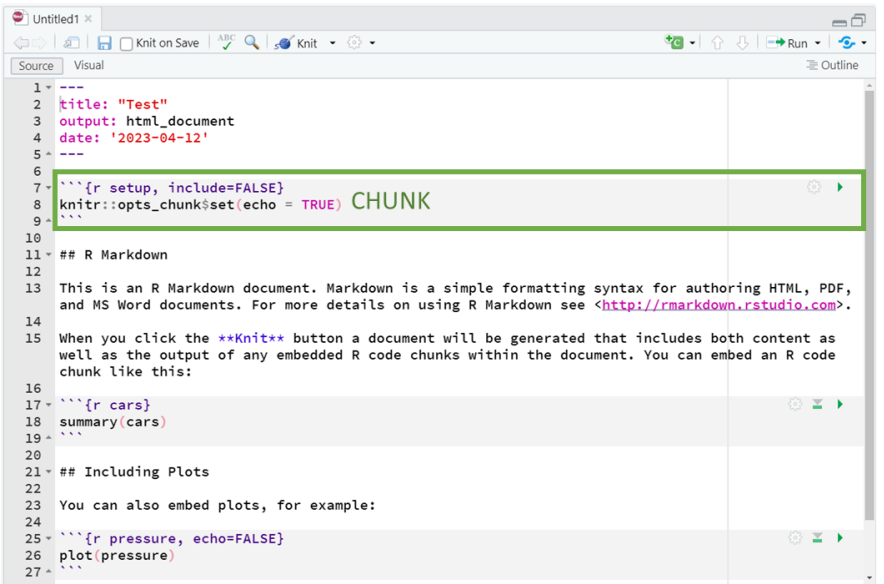


More options: Text in Rmd

1.6. Rmd chunk

The parts that are in grey are chunks.

- The code is written inside three inverted commas at the start and at the end and the r between {} This is because we need to tell which language are we using
- We can run the code using the green arrow that looks like a ▶, using the button **Run** in the upper part of the code editor, or using **ctrl+enter**.
- The results in Rmd appears in the code editor document, not in the console.



1.7. Rmd chunk

We can add new code chunks using **Ctrl+Alt+I** , the **back ticks** or in the green square with a C in the upper part of the code editor **+**.

The R code needs to be inside the chunks (the shadowed grey parts) to run.

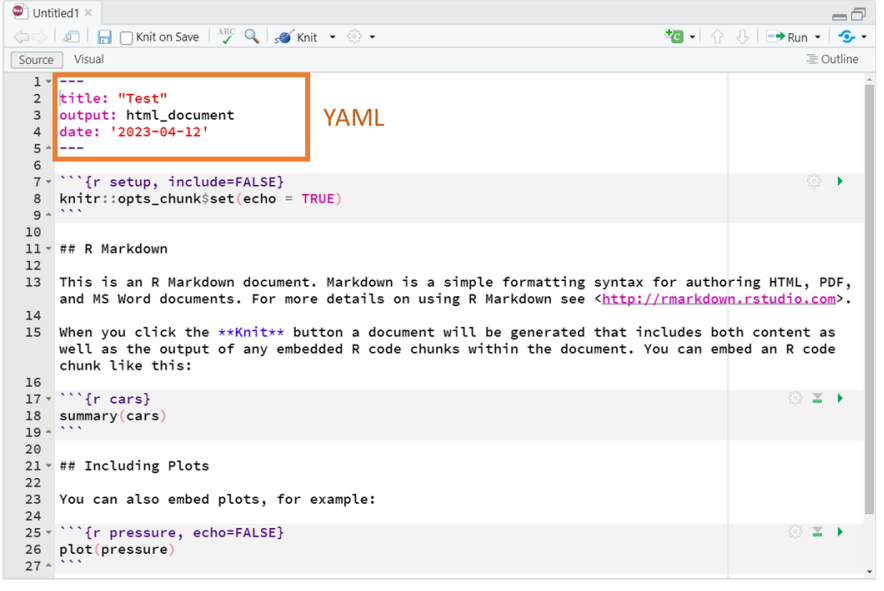
Common mistakes are:

- Not closing the parenthesis **{}**
- Not having the three inverted commas
- Add code outside the codechunks

1.8. YAML

YAML means *"YAML Yet Ain't Markup Language"*.

By default it will show: title, author, date, **output**.

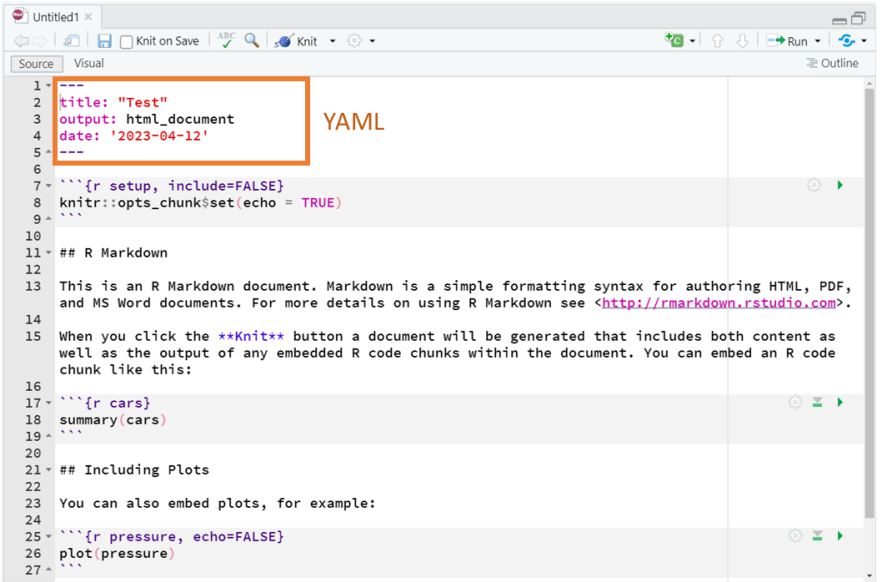


1.8. YAML

When you change the YAML, the information that appears in your report will change
If you chnge the output, different type of reports will be generated.

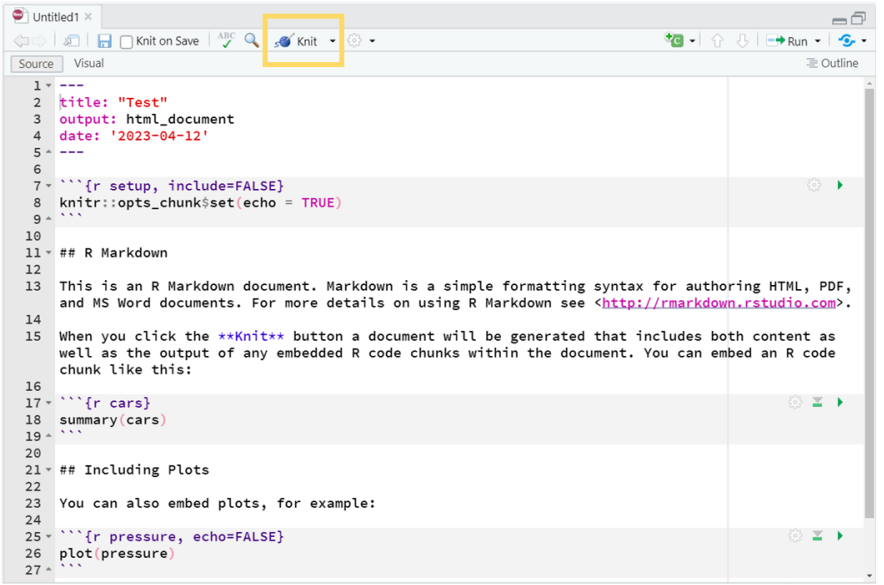
Common mistakes

When you **knit** or render your report, there is something incompatible with Pandoc (or LateX)



1.9. Knitr

There is a botton that says **knit** with a knitting blue ball. This bottons generates the report.



Note: every time we knit the changes are saved on our document.

1.10. Rmd titles

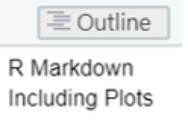
You can use the titles to navigate.

Search for the square with lines (say outline) or click on **Crtl+Shft+O**.

This can be very useful in large documents.

Main title use one hashtag #.

- First level #.
- Second level ##.
- Third level ###.



1.11. Rmd outputs

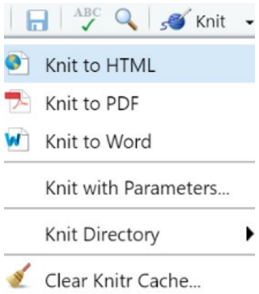
The documents can be exported to word, pdf and html.

Advantages and disadvantages for each one:

- word: easier to share and many journals ask for documents in this format
- pdf: easier to share but not easy to edit
- html: you can include a lot of type of contents ⭐, but might look unfamiliar

1.11. Rmd outputs

To change the output, you need to change it in the YAML, or chick on the arrow ▾ in the knit button and chose your format.



To modify the word format see here

Pause

- Open RStudio
- Open a Rmd file (File>NewFile>Rmd)
- Create three different formats: pdf, word and html
- Delete everything except for the YAML (change YAML to your info)
- Add a code chunk
- Add text

Example of code chunk

```
ingredients<-c('tomatoes','onions','pepper','salt','oil')
length(ingredients)
```

Note
Maybe you need to install tinytex

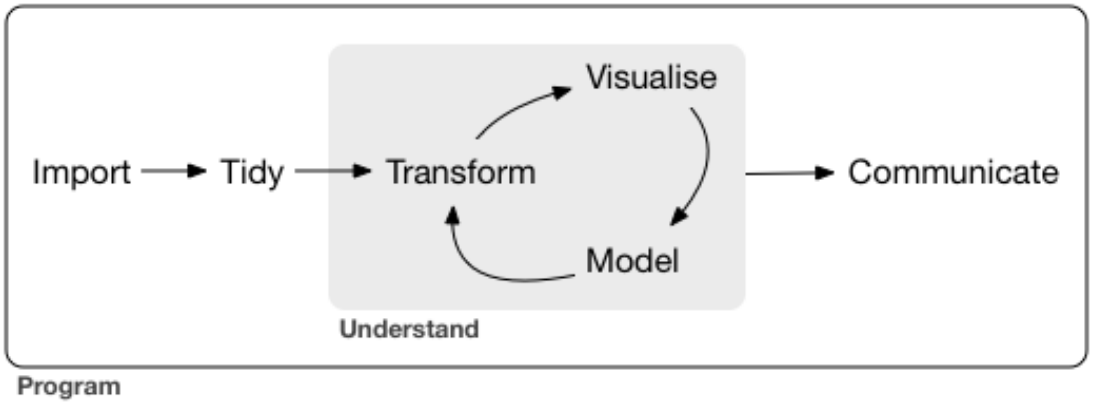
```
tinytex::install_tinytex()
```

- What is tiny text?


Load data
Bring the ingredients

2. Import

A typical R project looks like this:



Source: R4DS

2.1. Read files

To load data, we will use functions from the package tidyverse and the files:

- penguins1.csv
- penguins2.csv
- penguins3.txt
- penguins4.xlsx

Download here

Do you already have it installed?

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

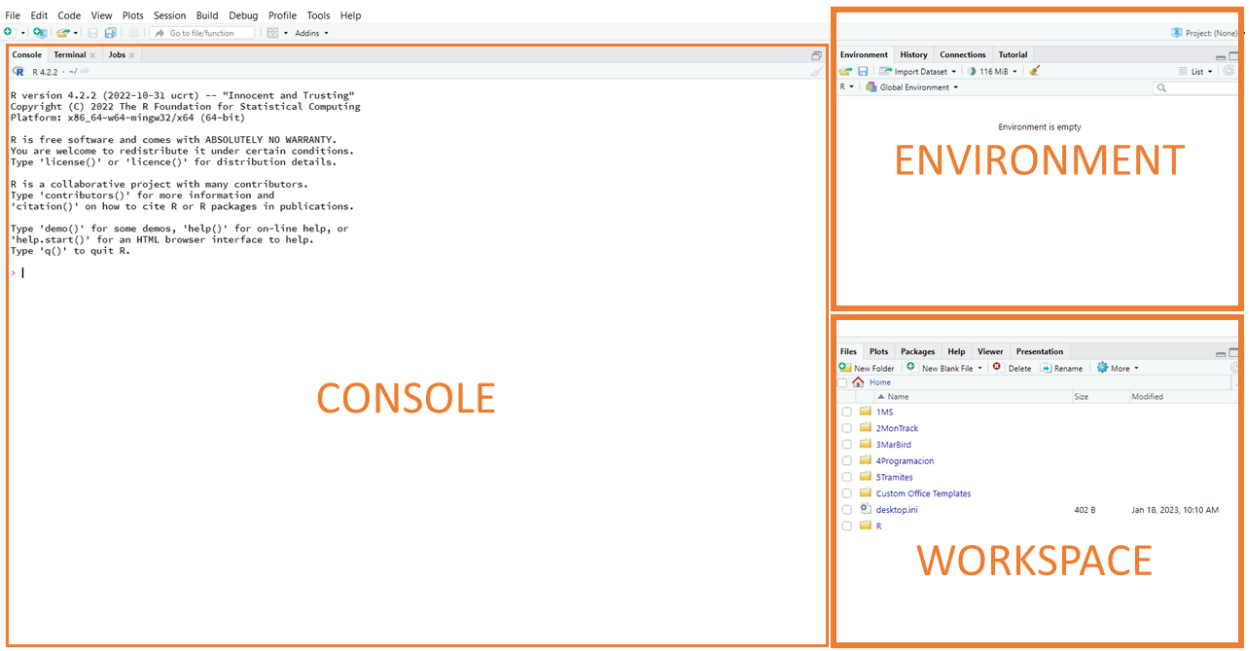
2.2. By hand

You can upload data by hand in your workspace

In the **environment** there is a part that says **Import Dataset**

Select the file **penguins1.csv**

Another option is to go to the **workspace** Files> Click on the file and **import data set**



2.3. csv format

Instead of clicks, we can write in the console or script:

```
penguins1<-read_csv("Downloads/penguins1.csv")
```

```
head(penguins1)
```

2.4. csv format

Now try opening **penguins2.csv**

This file instead of being separated by comas, its separated by colons ";"

Therefore, instead of using **read_csv** we will need to use **read_csv2**

To illustrate this issue, try loading the data using **read_csv**.

```
penguins2<-read_csv("Downloads/penguins2.csv")
```

```
head(penguins2)
```

Lets try now with **read_csv2**

```
penguins2<-read_csv2("Downloads/penguins2.csv")
```

```
head(penguins2)
```

2.5. Other formats

Click on the file **penguin3.txt**.

This one is separated by tabs.

read_tsv is for reading tab separated values.

```
penguins3<-read_tsv("Downloads/penguins3.txt")
```

```
head(penguins3)
```

2.6. Excel format

For loading excel data, there is a special package called **readxl**

```
library("readxl")
```

```
penguins4<- read_excel("Downloads/penguins4.xlsx")
```

```
head(penguins4)
```

2.7. From an url

Urls (Uniform Resource Locators) or links can also be source of data.

```
penguins5<- read_csv('https://github.com/MiriamLL/R_intro/blob/master/Do
```

Look at the first 5 rows of the data

```
head(penguins,5)
```

2.8. movebank

There is a package called move that can be use to access data stored in movebank.

To install:

```
install.packages('move')
```

```
library(move)
```

```
movebankLogin()
```

Add your login, if you have one, at the console.

2.9. movebank

It is more convenient to store your login information, but you have to be careful to not share the script with your login information.

```
loginStored <- movebankLogin(username="MiriamLerma", password="********"
```

```
my_study<-'FTZ UCN Kelp Gull Chile'
```

```
MyGull<-getMovebankLocationData(study=my_study,
                                individual_local_identifier="KEGU-noband0
                                timestamp_start="202212010000000",
                                timestamp_end="20221205000000000",
                                sensorID="GPS")
```

It might give you a warning, but for now that is not important.

2.10. packages with data

Data can also be stored in packages.
For example palmerpenguins

```
install.packages("palmerpenguins")
```

```
library(palmerpenguins)
penguin6<-penguins
```

```
head(penguin6)
```

2.11. packages with data

Moreover, packages with data are not limited to data frames.

2.12. packages with data

For example, the package GermanNorthSea contains shapefiles

With 6 lines of code you can plot a map (Showing just for illustration purposes)

```
# install.packages("devtools")
devtools::install_github("MiriamL
```

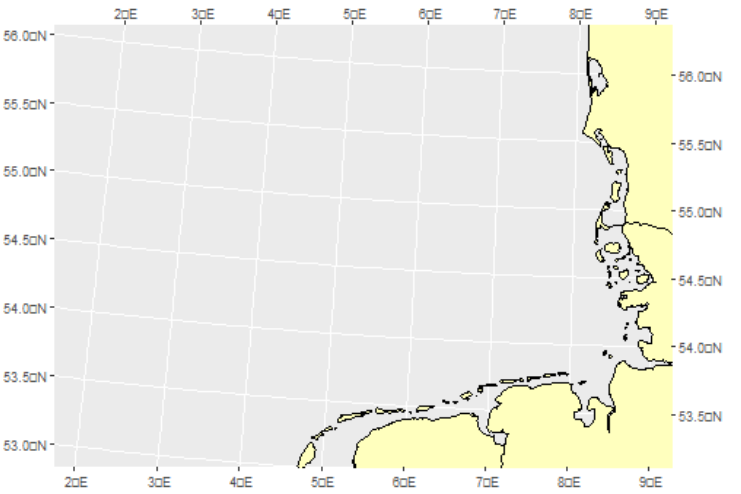
Now the package sf

```
#install.packages('sf')
library(sf)
library(ggplot2)
library(GermanNorthSea)
```

Load and plot some data

```
German_land<-GermanNorthSea::Germ
```

```
ggplot()+ geom_sf(data = German_l
  coord_sf(xlim = c(3790000,42500
                    label_axes =
```



Pause

Load penguin data

- Using read_csv
- Using read_csv2
- Using read_tsv
- Using read_excel

There are many other options of files.
Suggestions? here

Until here:

- index
- rmarkdown
- read files

Next part:

- basic operations
- columns and rows
- count
- distinct
- select
- filter
- mutate
- summarise
- drop_na
- join
- export
- contact

Operations

Kitchen utensils

3. Basic operations

Add

```
15+6
```

```
## [1] 21
```

Subtract

```
4-6
```

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

3. Basic operations

Divide

```
1700/8
```

```
## [1] 212.5
```

Multiply

```
20*20
```

```
## [1] 400
```

3.1. Using objects

¿ 🍕 = 😃 ?

How many people are here today?

```
People<-4+5+1
Pizza<-5*8
```

How many pieces each one gets?

```
Pizza/People
```

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

3.2. Using objects

Mean

```
cooking_temp<-c(134,145,167,200)
mean(cooking_temp)
```

```
## [1] 161.5
```

Median

```
median(cooking_temp)
```

```
## [1] 156
```

Standard deviation

```
sd(cooking_temp)
```

```
## [1] 29.10326
```

3.2. Using objects

Range

```
range(cooking_temp)
```

```
## [1] 134 200
```

Minimum

```
min(cooking_temp)
```

```
## [1] 134
```

Maximum

```
max(cooking_temp)
```

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

3.3. Look for help

```
mean(1,3,6,9,12)
```

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

Why 1? That can't be

Ask for help using **?**

```
?mean
```

The instructions will appear in the **workspace**, in the **Help** section

See in the examples, they all have a c from ***concatenate***

```
mean(c(1,3,6,9,12))
```

```
[1] 6.2
```

Now is working!

3.3. Look for help

One of the strenghts of R is that is widely used and there is a lot of webpages to search for help.

Be patience, check if you make a typo and if not copy and paste the error.

Reliable sources:

- stackoverflow
- mastodon (before was twitter but the r community move to this platform) Hashtags: #rstats

Pause

Practice ✏️

- For each salad 🥗 I need 1.3 cucumbers, how many cucumbers should I buy?

```
Salat<-3
Cucumber<-2
Salat*Cucumber
```

- I also want to make some cakes 🎂 and I need 200 g of sugar. How many grams of sugar do I need for preparing 5 cakes?

```
Cakes<-5
Sugar<-200
```

Until here:

- index
- rmarkdown
- read files
- basic operations

Next part:

- columns and rows
- count
- select
- filter
- mutate
- summarise
- unique
- drop_na
- join
- export
- contact


Data frames
Recipe book
57

4.1. Inspect data frames

Load data

```
library(palmerpenguins)
```

```
penguins<-penguins
```

Check first 5 rows

```
head(penguins,5)
```

Check last 5 rows

```
tail(penguins,5)
```

4.3. Rows

When you want to inspect specific rows, rows number is writen at the **first** position.

```
(penguins[1,])
```

Check first 3 rows.
The : is as "from A to B".

```
(penguins[1:3,])
```

4.4. Columns

The columns go on the **second** position.

```
head(penguins[,1])
```

Another way to do it is with the $ and the column name.

```
head(penguins$species)
```

4.5. Column and row

Look for a specific value [row, column]

```
(penguins[1,1])
```

```
## # A tibble: 1 × 1
##   species
##   <fct>
## 1 Adelie
```

```
(penguins[3,2])
```

```
## # A tibble: 1 × 1
##   island
##   <fct>
## 1 Torgersen
```


Tidy data
Ready for cooking

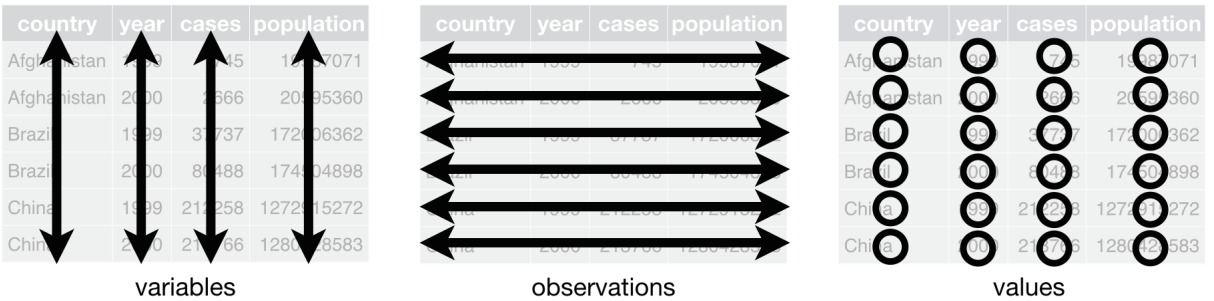
5.1. Tidy data

Data wrangling is usually the longest and slowest process and you can expect to do this several times.

Tidy data is a data structure to facilitate the analyses.

There are three interrelated rules which make a dataset tidy:

- Each variable must have its own column.
- Each observation must have its own row.
- Each value must have its own cell.



5.2. Recomendations

- To reduce the time organizing your data, is important to think earlier how are you going to collect and store your data. 🕐

Why to use tidydata?

- Many commands will assume that your data is organized.
- Is the expected format for statistical analyses.
- Its easier to plot organized data.
- When sharing the data it would be easier to understand.

Pause

Practice ✎

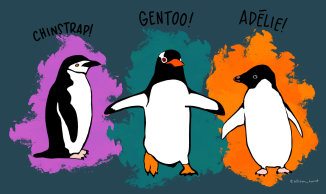
- Calculate the range of the body mass of the penguins.

```
range(penguins$body_mass_g,
      na.rm=TRUE)
```

- Calculate the mean of the body mass of the penguins.

```
mean(penguins$body_mass_g,
     na.rm=TRUE)
```

Note that na.rm allows you to ignore NAs



Until here:

- index
- rmarkdown
- read files
- basic operations

Next part:

- columns and rows
- count
- select
- filter
- mutate
- summarise
- unique
- drop_na
- join
- export
- contact


Functions
Cooking

6. Tidyverse

There are many ways to organize and wrangle your data.

Here we will cover those from the **tidyverse**.

```
library(tidyverse)
```

Tidyverse include many packages, a lot of them are specific for inspect and **wrangle** your data.

6.1. Pipe

A **pipe** is an argument we will use very often.
The **pipe** allows to chain several functions.

In your keyboard: strg+alt+M

```
%>%
```

6.2. count()

This functions lets you quickly count the unique values of one or more variables

Load library

```
library(tidyverse)
```

Sample size?

```
penguins %>%
  count()
```

Sample size per species?

```
penguins %>%
  count(species)
```

Per island and per species?

```
penguins %>%
  count(island,species)
```

6.3. unique() or distinct()

Allows you to see unique values or factors.

Using base R

```
unique(penguins$species)
```

Using tidyverse

```
penguins %>%
  distinct(species)
```

6.4. select()

Select variables in a data frame

Select one column

```
penguins %>%
  select(species)
```

Remove one column using " - "

```
penguins %>%
  select(-sex)
```

Select all columns expect this one using " ! "

```
penguins %>%
  select(!sex)
```

Select columns in between using ": "

```
penguins %>%
  select(bill_length_mm:body_mass
```

Using the final letter of the string

```
penguins %>%
  select(ends_with("mm"))
```

Using the first letters of the string

```
penguins %>%
  select(starts_with("bill"))
```

6.5. filter()

The filter() function is used to subset a data frame, retaining all rows that satisfy your conditions.

There are many functions and operators, some useful expressions are:

- The symbol **==** means 'same as'
- The symbol **!=** means 'not the same as'
- The symbol **>** means 'larger than'
- The symbol **<** means 'smaller than'
- The symbol **>=** means 'larger or same as'
- The symbol **<=** means 'smaller or same as'
- The symbol **&** means 'and'
- The symbol **|** means 'or'

6.5. filter(==)

- The symbol == means 'same as'

```
penguins %>%
  filter(sex == 'female')
```

Note variables are without quotes and observations in quotes.

Is there actually something different in the object at your environment?

To change the object we need to create a new data frame.

```
female_penguins<-penguins %>%
  filter(sex == 'female')
```

6.5. filter(<=)

- The symbol **<=** means 'smaller or same as'

```
penguins %>%
  filter(bill_lenght_mm <= 39.1)
```

- The symbol **>=** means 'larger or same as'

```
penguins %>%
  filter(bill_length_mm >= 39.1)
```

- The symbol **&** means 'and'

```
penguins %>%
  filter(island == 'Biscoe' & species =='Adelie')
```

6.6. mutate()

mutate() creates new columns that are functions of existing variables. It can also modify (if the name is the same as an existing column) and delete columns (by setting their value to NULL).

```
penguins<-penguins %>%
  mutate(body_mass_kg = body_mass_g / 1000)
```

6.7. group_by() y summarise()

group_by() lets you select an specific column for grouping the factors within summarise() can be used to use specific operations for each factor defined in the group_by

```
penguins %>%
  group_by(year) %>%
  summarise(mean_bill_length=mean(bill_length_mm))
```

```
## # A tibble: 3 × 2
##    year mean_bill_length
##   <int>            <dbl>
## 1  2007             NA
## 2  2008             43.5
## 3  2009             NA
```

6.8. drop_na

This functions allows you to ignore or remove NAs

```
penguins %>%
  drop_na(bill_length_mm)
```

Another option is to remove the nas

```
clean_penguins <- penguins %>%
  filter(!is.na(bill_length_mm))
```

6.9. lubridate

We often use date and time, so lets try with an example using this data type.

The package **lubridate** provides tools that make it easier to parse and manipulate dates.

```
library(lubridate)
```

```
ymd_hms("2010-12-13 15:30:30")
```

You can **extract** some elements from dates and times

```
ymd_hms("2010-12-13 15:30:30") %>%
  month()
```

```
## [1] 12
```

6.9. lubridate

Lets try with this data frame.

```
my_timestamps<-data.frame(timestamp=c("2010-12-13 13:30:30","2010-12-13
```

Using mutate we can separate elements form the date and time

```
my_timestamps %>%
  mutate(
    my_hours = hour(timestamp),
    my_minutes = minute(timestamp),
    my_seconds = second(timestamp)
  )
```

Pause

Practice ✎

```
penguins %>%
  count()
```

```
penguins %>%
  select(especie)
```

```
penguins %>%
  group_by(species,sex) %>%
  drop_na(body_mass_g)%>%
  summarise(mean_body_mass_g = me
  mutate(mean_body_mass_kg = mean
```

Until here:

- index
- rmarkdown
- read files
- basic operations
- columns and rows
- count
- distinct
- select
- filter
- mutate
- summarise
- drop_na

Next part:

- join
- export
- contact


Join
Mixing ingredients

7. Mutating joins

Mutating joins add columns from y to x, matching observations based on the keys. There are four mutating joins: the inner join, and the three outer joins.

Lets create a new data set.

```
bird_id<-c("ID01","ID02","ID03","ID04","ID05",
      "ID06","ID07","ID08","ID09","ID10")
bird_mass<-c(1.5,2.0,3.5,4.1,2.6,3.7,8.9,2.5,6.3,1.0)
bird_gps<-c(50010,50020,50035,50001,50006,50003,50008,50002,50003,50001)
```

We might have two data sets

On one hand, the measurement data...

```
bird_measurements<-
  data.frame(bird_id,
             bird_mass)
```

... on the other, field data.

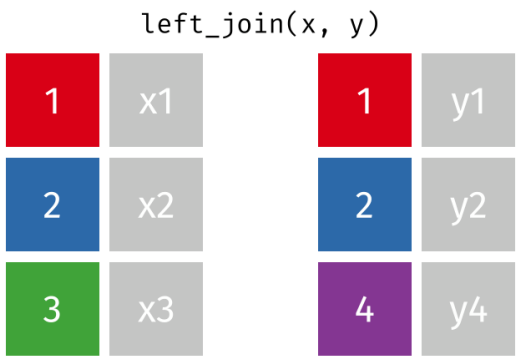
```
bird_tracking <-
  data.frame(bird_id,
             bird_gps)
```

7.1. left_join()

To join them we can use the function **left_join()**
But it is important to have a **key** to match the observations

```
bird_joined<-left_join(bird_measurements,
                       bird_tracking,
                       by = "bird_id")
```

left_join() uses the **key** to join the data frames



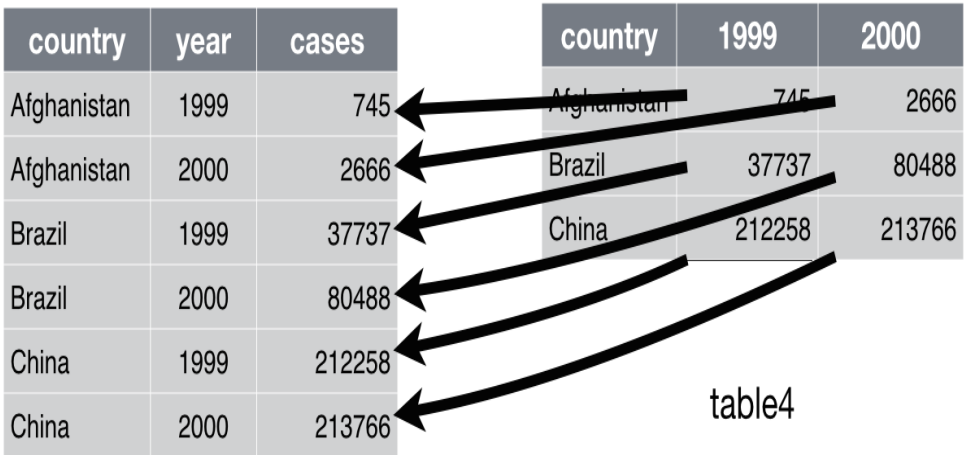
Other options

7.2. pivot_longer

Sometimes the data are not organized in a way that each observation has a row and a column.

This is very common, particularly in the lab or fieldwork because is not the same how we write in a notebook than in the computer.

To re-organized, we can use the function **pivot_longer**.



7.2. pivot_longer

Lets imagine we have data from five species and their number of locations among three different years.

```
bird_id<-c("ID01","ID02","ID03","ID04","ID05",
      "ID06","ID07","ID08","ID09","ID10")
year_2010<-c(5,4,5,6,7,3,2,1,9,10)
year_2011<-c(3,2,1,9,4,5,6,7,3,2)
year_2012<-c(6,2,3,7,8,2,1,9,4,5)
```

New data frame

```
bird_nlocs<-data.frame(bird_id,year_2010,year_2011,year_2012)
```

```
head(bird_nlocs,5)
```

```
##   bird_id year_2010 year_2011 year_2012
## 1    ID01         5         3         6
## 2    ID02         4         2         2
## 3    ID03         5         1         3
## 4    ID04         6         9         7
## 5    ID05         7         4         8
```

7.2. pivot_longer

pivot_longer "lengthens" data, increasing the number of rows and decreasing the number of columns.

```
bird_long <- bird_nlocs %>%
  pivot_longer(c(year_2010,year_2011,year_2012),
    names_to = "year",
    values_to = "nlocs" )
```

```
head(bird_long,5)
```

```
## # A tibble: 5 × 3
##   bird_id year      nlocs
##   <chr>   <chr>     <dbl>
## 1 ID01    year_2010     5
## 2 ID01    year_2011     3
## 3 ID01    year_2012     6
## 4 ID02    year_2010     4
## 5 ID02    year_2011     2
```

7.3. pivot_wider

The opposite will be to separate the columns.

pivot_wider() "widens" data, increasing the number of columns and decreasing the number of rows.

country	year	key	value
Afghanistan	1999	cases	745
Afghanistan	1999	population	19987071
Afghanistan	2000	cases	2666
Afghanistan	2000	population	20595360
Brazil	1999	cases	37737
Brazil	1999	population	172006362
Brazil	2000	cases	80488
Brazil	2000	population	174504898
China	1999	cases	212258
China	1999	population	1272915272
China	2000	cases	213766
China	2000	population	1280428583

table2

country	year	cases	population
Afghanistan	1999	745	19987071
Afghanistan	2000	2666	20595360
Brazil	1999	37737	172006362
Brazil	2000	80488	174504898
China	1999	212258	1272915272
China	2000	213766	1280428583

The most important arguments are **names_from** which are going to be the names of the columns created after (often the column with factors) and **values_from** is the the name of the column with the values (often the columns with numbers)

7.4. paste or unite

The argument **paste** or **paste0** from base R allows you to paste together multiple columns

```
bird_long$unique_id<-paste0(bird_long$bird_id,'_',bird_long$year)
```

The argument **unite** is similar, but lets you to paste together multiple columns into one.

```
bird_long<-bird_long %>%
  unite(col = unique_id2,
        c("bird_id", "year"),
        sep = "_",
        remove=FALSE)
```

Note it will get rid of the original column, so if you don't want to eliminate the original column add **remove = FALSE.**

7.5. separate

The argument **separate** allows you to separate the values from one column into two columns.

```
bird_long %>%
  separate(col = unique_id,
           into = c("id", "text","year"),
           sep = "_")
```

Note it will get rid of the original column, so if you don't want to eliminate the original column add **remove = FALSE**.

```
bird_long<-bird_long %>%
  separate(col = unique_id,
           into = c("id", "text","year"),
           sep = "_",
           remove = FALSE)
```

7.6. rename

The argument **rename** allows to change the name of one or several columns.
The new name is written first and the old name comes after.

An example changing the name of one column

```
bird_long %>%
  rename(unique_identifier = unique_id2)
```

7.7. relocate

The argument **relocate** allows you to reorganize your columns and keeping just those that you are interested on.

```
bird_long %>%
  relocate(bird_id,year,nlocs)
```

Using this argument together with select you can keep only the columns of interest.

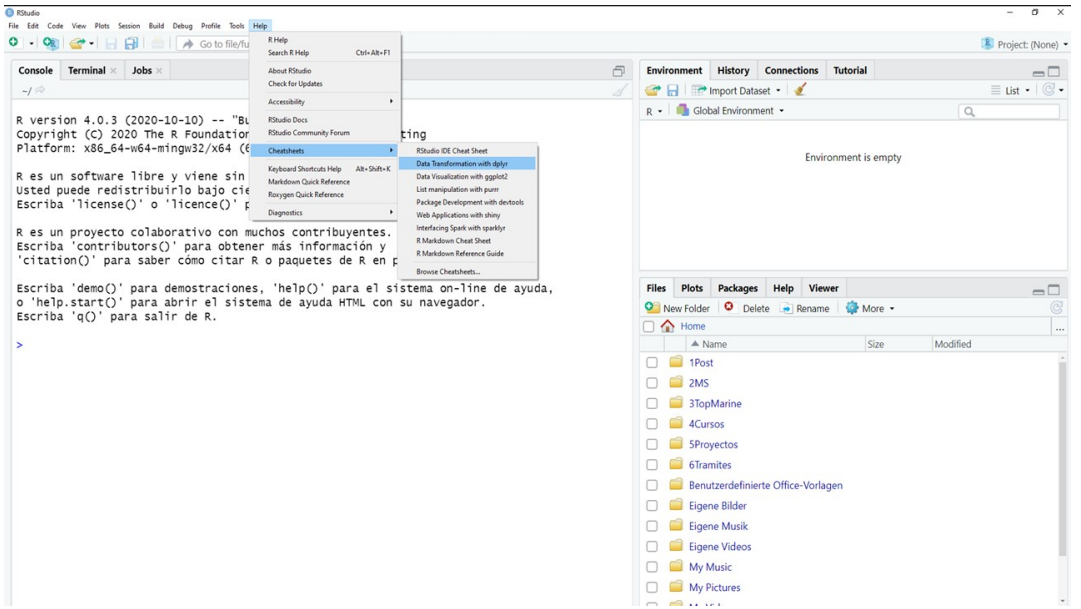
```
bird_long %>%
  select(bird_id,year,nlocs)%>%
  relocate(bird_id,year,nlocs)
```

7.8. keep learning

Use the dplyr cheatSheet.
Try the exercises from Allison horst.

Cheatsheets

Help > Cheat sheet > Data transformation with dplyr



Pause

Practice

```
left_join(bird_measurements,
          bird_tracking,
          by = "bird_id")
```

```
bird_long <- bird_nlocs %>%
  pivot_longer(c(year_2010,year_2
    names_to = "year",
    values_to = "nlocs" )
```

```
bird_wide<-bird_long %>%
  pivot_wider(names_from = year,
              values_from = nlocs
```

Until here:

- index
- rmarkdown
- read files
- basic operations
- bolumns and Rows
- tidydata
- count
- distinct
- select
- filter
- mutate
- summarise
- drop_na
- join

Next part:

- export
- contact


Export
Storing in order

8. Export

Similar to the read files arguments (read_csv), each one has their equivalent to write.

- write_csv()
- write_csv2()
- write_tsv()
- write_delim()

Pause

Practice ✎

Define a folder

```
library(here)
ResultsFolder<-here()
```

Export to csv

```
write_csv(
  bird_joined,
  file =paste0(ResultsFolder,'/bird_joined.csv'))
```

Back to

- rmarkdown
- read files
- basic operations
- columns and rows
- tidydata
- distinct
- count
- select
- filter
- mutate
- summarise
- drop_na
- join
- export

Contact

This materials are free of use
Download the presentation here: github and webpage

Home

Index