

20 Things to Know About R

Alexandre Courtiol

Leibniz Institute of Zoo and Wildlife Research

September 2022



**Leibniz Institute for Zoo
and Wildlife Research**
IN THE FORSCHUNGSVERBUND BERLIN E.V.



20 Things to Know About R

- 1 What are **R** and RStudio?
- 2 How to best setup **R** & RStudio?
- 3 How to best organise your work?
- 4 How to write reproducible code?
- 5 How to use **R** as a simple calculator?
- 6 How to store the results of operations?
- 7 How to do things beyond simple algebra?
- 8 How to use a new function?
- 9 How to decipher complex function calls?
- 10 How to import data in **R**?
- 11 How to manipulate data frames?
- 12 How to manipulate lists?
- 13 How to manipulate vectors?
- 14 How to use packages?
- 15 How to use `{dplyr}` for an efficient manipulation of data frames?
- 16 How to plot things?
- 17 How to use `{ggplot2}` for an efficient production of plots?
- 18 When to update what?
- 19 How to reach the level A2/B1?
- 20 How to reach levels beyond B1?

1. What are **R** and RStudio?

Why learning **R**?

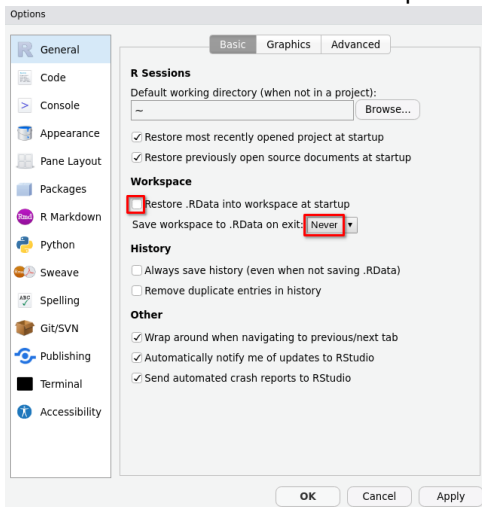
- best software out there for most data science tasks
- open source and free
- possible to learn for people with no background in computer science
- rich in functionality with close to 100,000 free packages
(<https://rdr.io>; <https://www.rdocumentation.org>)
- friendly large community
(twitter [#rstats](#), <https://rfordatascience.slack.com>, <https://community.rstudio.com>, gatherings)

RStudio is a GUI to **R** that is fancier and more modern than the one provided by default

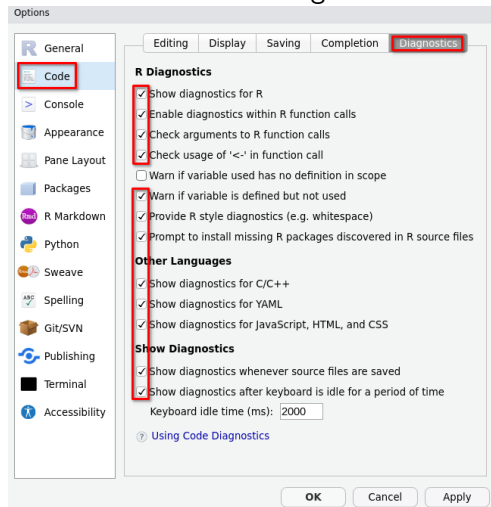
2. How to best setup R & RStudio?

Open: Menu / Tools / Global options and set things as follows:

Never save or restore the workspace!



Activate code diagnostics



3. How to best organise your work?

Before you start some new work in **R**,

- ❶ create a new RStudio project (Menu / File / New Project... / ...)
- ❷ create a new **R** Script file (Menu / File / New File / R Script)
- ❸ save the created **R** Script file into the project folder directory

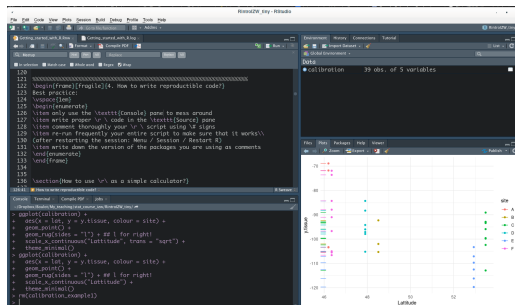
NB:

- an RStudio project is a folder containing the different files for a given project, including a simple project file that allows you to open RStudio with the correct working directory set
- an **R** Script is a (text) file where you can write **R** code
- alternatives to **R** Scripts exist to implement your work if you need to format text around your code (e.g. **R** Markdown files)

4. How to write reproducible code?

Best practice:

- 1 only use the Console pane to mess around
- 2 write proper **R** code in the Source pane
- 3 comment thoroughly your **R** script using `#` signs
- 4 re-run frequently your entire script to make sure that it works
(after restarting the session: Menu / Session / Restart R)
- 5 write down the version of the packages you are using as comments



5. How to use R as a simple calculator?

R can perform basic arithmetic:

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

1 - 1
## [1] 0

2 * pi
## [1] 6.283185

3 / 2
## [1] 1.5

5^2
## [1] 25

5^(2 + 1)
## [1] 125

Inf/Inf
## [1] NaN
```

5. How to use R as a simple calculator?

R can perform basic arithmetic:

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

1 - 1
## [1] 0

2 * pi
## [1] 6.283185

3 / 2
## [1] 1.5

5^2
## [1] 25

5^(2 + 1)
## [1] 125

Inf/Inf
## [1] NaN
```

R can perform logical operations:

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

(1 == 1) & (1 == 2)
## [1] FALSE

(1 == 1) | (1 == 2)
## [1] TRUE

1 != 2
## [1] TRUE

!(1 == 2)
## [1] TRUE

2 >= 1
## [1] TRUE

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


5. How to use R as a simple calculator?

R can perform basic arithmetic:

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

1 - 1
## [1] 0

2 * pi
## [1] 6.283185

3 / 2
## [1] 1.5

5^2
## [1] 25

5^(2 + 1)
## [1] 125

Inf/Inf
## [1] NaN
```

R can perform logical operations:

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

(1 == 1) & (1 == 2)
## [1] FALSE

(1 == 1) | (1 == 2)
## [1] TRUE

1 != 2
## [1] TRUE

!(1 == 2)
## [1] TRUE

2 >= 1
## [1] TRUE

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

As for most other programming languages, avoid equality tests for floating-point numbers!

```
0.8 - 0.3 - 0.5 == 0.8 - 0.5 - 0.3 ## some functions should be used instead
## [1] FALSE
```

Compute $\sqrt{\frac{2^{3+1}}{4} - 20}$

NB: $\sqrt{x} = x^{\frac{1}{2}}$

6. How to store the results of operations?

The results of operations are stored into *objects* that are created using the “arrow” assignment operator:

```
one_plus_one <- 1 + 1 ## storing the result
```

6. How to store the results of operations?

The results of operations are stored into *objects* that are created using the “arrow” assignment operator:

```
one_plus_one <- 1 + 1 ## storing the result
```

Objects can then be used through their name (that is the whole point):

```
one_plus_one ## displaying the result
## [1] 2
one_plus_one_plus_one <- one_plus_one + 1
one_plus_one_plus_one
## [1] 3
```

6. How to store the results of operations?

The results of operations are stored into *objects* that are created using the “arrow” assignment operator:

```
one_plus_one <- 1 + 1 ## storing the result
```

Objects can then be used through their name (that is the whole point):

```
one_plus_one ## displaying the result
## [1] 2
one_plus_one_plus_one <- one_plus_one + 1
one_plus_one_plus_one
## [1] 3
```

Tips:

```
(one_times_two <- 1 * 2) ## storing and displaying the result at once
## [1] 2
```

6. How to store the results of operations?

The results of operations are stored into *objects* that are created using the “arrow” assignment operator:

```
one_plus_one <- 1 + 1 ## storing the result
```

Objects can then be used through their name (that is the whole point):

```
one_plus_one ## displaying the result
## [1] 2
one_plus_one_plus_one <- one_plus_one + 1
one_plus_one_plus_one
## [1] 3
```

Tips:

```
(one_times_two <- 1 * 2) ## storing and displaying the result at once
## [1] 2
```

- `->` works too (if you switch the left hand side and the right hand side)
- “`_`” and “`.`” are OK but avoid spaces & other weird characters in names
- names are case sensitive

7. How to do things beyond simple algebra?

For doing more complex things, one has to use *functions*:

```
vector_x <- c(1, 4, 10)
```

```
mean_x <- mean(x = vector_x)
mean_x
## [1] 5
```

```
sd_x <- sd(x = vector_x)
sd_x
## [1] 4.582576
```

```
cv_x <- sd_x/mean_x
cv_x
## [1] 0.9165151
```

```
round(x = cv_x, digits = 2)
## [1] 0.92
```

Note: all what is written in red above are functions

8. How to use a new function?

Before using a new function, it is a good idea to check its documentation:

```
mean()
```

```
?mean()
```

Usage:

```
mean(x, ...)  
## Default S3 method:  
mean(x, trim = 0, na.rm = FALSE, ...)
```

Arguments:

- `x`: An R object. Currently there are methods for numeric/logical vectors and date, date-time and time interval objects, and for data frames all of whose columns have a method. Complex vectors are allowed for 'trim = 0', only.
- `trim`: the fraction (0 to 0.5) of observations to be trimmed from each end of 'x' before the mean is computed. Values of trim outside that range are taken as the nearest endpoint.
- `na.rm`: a logical value indicating whether 'NA' values should be stripped before the computation proceeds.

[...]

9. How to decipher complex function calls?

In **R**, it is easy to write synonymous code that *looks* very different:

Example 1:

```
round(mean(c(1, 4, 10))/sd(c(1, 4, 10)), 2)
## [1] 1.09
```

9. How to decipher complex function calls?

In **R**, it is easy to write synonymous code that *looks* very different:

Example 1:

```
round(mean(c(1, 4, 10))/sd(c(1, 4, 10)), 2)
## [1] 1.09
```

Example 2:

```
c(1, 4, 10) |>
  mean() |>
  prod(1/sd(c(1, 4, 10))) |>
  round(2)
## [1] 1.09
```

9. How to decipher complex function calls?

In **R**, it is easy to write synonymous code that *looks* very different:

Example 1:

```
round(mean(c(1, 4, 10))/sd(c(1, 4, 10)), 2)
## [1] 1.09
```

Example 2:

```
c(1, 4, 10) |>
  mean() |>
  prod(1/sd(c(1, 4, 10))) |>
  round(2)
## [1] 1.09
```

- To decipher the first example, run things from the inside out, step-by-step
- To decipher the second example, run things adding one line at a time (always skipping the final pipe)

Here the implementation from Example 1 is easier to read, but this is not always the most natural way to write

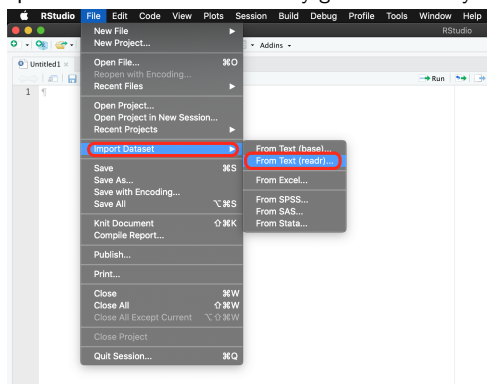
10. How to import data in R?

R can read (and write) many formats storing data, such as:

- tabulated text files (*.csv, *.txt, ...) → no pkg or {readr}
- MS Excel files (*.xlsx) → {readxl}
- binary R files (*.rda, *.RData, *.rds) → no pkg

For importing data you can often simply rely on the GUI

(but for reproducibility do copy and paste the R code automatically generated into your script!):



11a. How to manipulate data frames?

The most common class of objects used for storing data in **R** is the data frame!

Example: the `calibration_example1` dataset

```
library(readr)
calibration <- read_csv("data/calibration_example1.csv")
calibration
## # A tibble: 39 x 5
##   site    lat  long y.tissue sd.y.tissue
##   <chr> <dbl> <dbl>   <dbl>     <dbl>
## 1 A      46.2  18.9   -71.8         1
## 2 A      46.2  18.9  -103.         1
## 3 A      46.2  18.9   -68.9         1
## # ... with 36 more rows
```

- each column is a variable, usually corresponding to a vector
(a series of elements of 1 type)
- all columns have the same length (rectangular format)
- contains column names (usually informative) and row names (usually not informative)
- a tibble is one of the possible formats for data frames, but all formats work similarly

11b. How to manipulate data frames?

```
calibration$y.tissue ## extract vector
## [1] -71.8 -103.4 -68.9 -103.3 -92.3 -102.0 -105.3 -93.4 -95.9 -93.2 -93.2 -92.9 -99.8
## [14] -89.1 -112.9 -99.6 -102.4 -105.2 -97.9 -97.1 -96.0 -88.3 -94.2 -84.3 -112.5 -109.7
## [27] -103.3 -116.2 -107.1 -119.7 -96.7 -71.9 -106.6 -105.5 -105.0 -84.3 -87.9 -91.0 -73.6
```

11b. How to manipulate data frames?

```
calibration$y.tissue ## extract vector
## [1] -71.8 -103.4 -68.9 -103.3 -92.3 -102.0 -105.3 -93.4 -95.9 -93.2 -93.2 -92.9 -99.8
## [14] -89.1 -112.9 -99.6 -102.4 -105.2 -97.9 -97.1 -96.0 -88.3 -94.2 -84.3 -112.5 -109.7
## [27] -103.3 -116.2 -107.1 -119.7 -96.7 -71.9 -106.6 -105.5 -105.0 -84.3 -87.9 -91.0 -73.6

calibration[calibration$site == "B", c("lat", "long", "y.tissue")] ## filter rows and columns
## # A tibble: 3 x 3
##   lat long y.tissue
##   <dbl> <dbl>   <dbl>
## 1  48.5  19.1   -92.3
## 2  48.5  19.1   -102
## 3  48.5  19.1   -105.
```

11b. How to manipulate data frames?

```
calibration$y.tissue ## extract vector
## [1] -71.8 -103.4 -68.9 -103.3 -92.3 -102.0 -105.3 -93.4 -95.9 -93.2 -93.2 -92.9 -99.8
## [14] -89.1 -112.9 -99.6 -102.4 -105.2 -97.9 -97.1 -96.0 -88.3 -94.2 -84.3 -112.5 -109.7
## [27] -103.3 -116.2 -107.1 -119.7 -96.7 -71.9 -106.6 -105.5 -105.0 -84.3 -87.9 -91.0 -73.6
```

```
calibration[calibration$site == "B", c("lat", "long", "y.tissue")] ## filter rows and columns
## # A tibble: 3 x 3
##   lat long y.tissue
##   <dbl> <dbl>   <dbl>
## 1  48.5  19.1   -92.3
## 2  48.5  19.1   -102
## 3  48.5  19.1   -105.
```

```
calibration$sd.y.tissue <- NULL ## delete a column
calibration$species <- "my species" ## add a column
calibration
## # A tibble: 39 x 5
##   site lat long y.tissue species
##   <chr> <dbl> <dbl>   <dbl> <chr>
## 1 A    46.2  18.9   -71.8 my species
## 2 A    46.2  18.9   -103. my species
## 3 A    46.2  18.9   -68.9 my species
## # ... with 36 more rows
```


12. How to manipulate lists?

Most functions doing something more complicated than simple arithmetic produce lists:

```
calib2sp <- calibration[calibration$site %in% c("A", "C"), ]  
test_calib2sp <- t.test(y.tissue ~ lat, data = calib2sp)
```

12. How to manipulate lists?

Most functions doing something more complicated than simple arithmetic produce lists:

```
calib2sp <- calibration[calibration$site %in% c("A", "C"), ]  
test_calib2sp <- t.test(y.tissue ~ lat, data = calib2sp)
```

```
str(test_calib2sp) ## reveals the (often hidden) structure of the list  
## List of 10  
## $ statistic : Named num 1.06  
## ..- attr(*, "names")= chr "t"  
## $ parameter : Named num 3.31  
## ..- attr(*, "names")= chr "df"  
## $ p.value : num 0.36  
## $ conf.int : num [1:2] -19.2 39.9  
## ..- attr(*, "conf.level")= num 0.95  
## $ estimate : Named num [1:2] -86.8 -97.2  
## ..- attr(*, "names")= chr [1:2] "mean in group 46.19747" "mean in group 53.483253"  
## $ null.value : Named num 0  
## ..- attr(*, "names")= chr "difference in means between group 46.19747 and group 53.483253"  
## $ stderr : num 9.79  
## $ alternative: chr "two.sided"  
## $ method : chr "Welch Two Sample t-test"  
## $ data.name : chr "y.tissue by lat"  
## - attr(*, "class")= chr "htest"
```

12. How to manipulate lists?

Most functions doing something more complicated than simple arithmetic produce lists:

```
calib2sp <- calibration[calibration$site %in% c("A", "C"), ]  
test_calib2sp <- t.test(y.tissue ~ lat, data = calib2sp)
```

```
str(test_calib2sp) ## reveals the (often hidden) structure of the list  
## List of 10  
## $ statistic : Named num 1.06  
## .. attr(*, "names")= chr "t"  
## $ parameter : Named num 3.31  
## .. attr(*, "names")= chr "df"  
## $ p.value : num 0.36  
## $ conf.int : num [1:2] -19.2 39.9  
## .. attr(*, "conf.level")= num 0.95  
## $ estimate : Named num [1:2] -86.8 -97.2  
## .. attr(*, "names")= chr [1:2] "mean in group 46.19747" "mean in group 53.483253"  
## $ null.value : Named num 0  
## .. attr(*, "names")= chr "difference in means between group 46.19747 and group 53.483253"  
## $ stderr : num 9.79  
## $ alternative: chr "two.sided"  
## $ method : chr "Welch Two Sample t-test"  
## $ data.name : chr "y.tissue by lat"  
## - attr(*, "class")= chr "htest"
```

You can extract named elements from lists as vectors in data frames:

```
test_calib2sp$p.value  
## [1] 0.3595509
```

You can extract elements from lists (named or not) using indexes too:

```
test_calib2sp[[3]]  
## [1] 0.3595509
```

13. How to manipulate vectors?

You can replace elements:

```
calibration$y.tissue[1] <- NA ## change is saved!
```

13. How to manipulate vectors?

You can replace elements:

```
calibration$y.tissue[1] <- NA ## change is saved!
```

You can select multiple elements:

```
calibration$y.tissue[1:10] ## change is not saved unless you assign it  
## [1]      NA -103.4  -68.9 -103.3  -92.3 -102.0 -105.3  -93.4  -95.9  -93.2
```

13. How to manipulate vectors?

You can replace elements:

```
calibration$y.tissue[1] <- NA ## change is saved!
```

You can select multiple elements:

```
calibration$y.tissue[1:10] ## change is not saved unless you assign it
## [1]      NA -103.4  -68.9 -103.3  -92.3 -102.0 -105.3  -93.4  -95.9  -93.2
```

You can select multiple using a vector of logicals:

```
calibration$site == "A"
## [1]  TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [17] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [33] FALSE FALSE FALSE FALSE FALSE FALSE FALSE
calibration$y.tissue[calibration$site == "A"]
## [1]      NA -103.4  -68.9 -103.3
```

13. How to manipulate vectors?

You can replace elements:

```
calibration$y.tissue[1] <- NA ## change is saved!
```

You can select multiple elements:

```
calibration$y.tissue[1:10] ## change is not saved unless you assign it
## [1]      NA -103.4  -68.9 -103.3  -92.3 -102.0 -105.3  -93.4  -95.9  -93.2
```

You can select multiple using a vector of logicals:

```
calibration$site == "A"
## [1]  TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [17] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [33] FALSE FALSE FALSE FALSE FALSE FALSE FALSE
calibration$y.tissue[calibration$site == "A"]
## [1]      NA -103.4  -68.9 -103.3
```

You can remove elements:

```
calibration$y.tissue[-2]
## [1]      NA  -68.9 -103.3  -92.3 -102.0 -105.3  -93.4  -95.9  -93.2  -93.2  -92.9  -99.8  -89.1
## [14] -112.9  -99.6 -102.4 -105.2  -97.9  -97.1  -96.0  -88.3  -94.2  -84.3 -112.5 -109.7 -103.3
## [27] -116.2 -107.1 -119.7  -96.7  -71.9 -106.6 -105.5 -105.0  -84.3  -87.9  -91.0  -73.6
```

14. How to use packages?

Packages extend **R** functionalities:

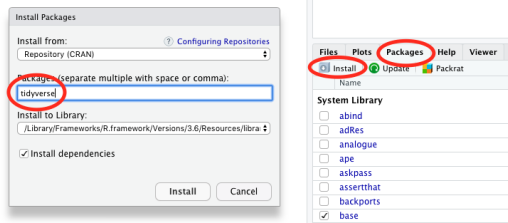
- for most users; e.g. `{dplyr}`, `{ggplot2}`
- for specific users; e.g. `{IsoriX}`, `{MixSIAR}`
- for developers creating packages; e.g. `{devtools}`, `{Rcpp}`

14. How to use packages?

Packages extend **R** functionalities:

- for most users; e.g. `{dplyr}`, `{ggplot2}`
- for specific users; e.g. `{IsoriX}`, `{MixSIAR}`
- for developers creating packages; e.g. `{devtools}`, `{Rcpp}`

Simple installation (once per **R** installation):

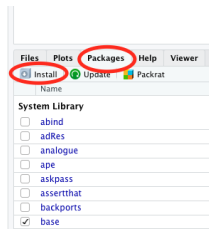
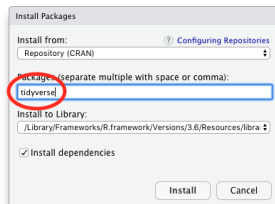


14. How to use packages?

Packages extend **R** functionalities:

- for most users; e.g. `{dplyr}`, `{ggplot2}`
- for specific users; e.g. `{IsoriX}`, `{MixSIAR}`
- for developers creating packages; e.g. `{devtools}`, `{Rcpp}`

Simple installation (once per **R** installation):



Loading (once per session):

```
library(dplyr)
```

15. How to use `{dplyr}` for an efficient manipulation of data frames?

You can all kind of data manipulation by combining 5 simple functions from `{dplyr}`:

- `select()` to keep or discard columns
- `group_by()` to define groups of rows for the following verbs
- `filter()` to keep or discard rows
- `mutate()` to create new columns
- `summarise()` to compute summary statistics

15. How to use {dplyr} for an efficient manipulation of data frames?

You can all kind of data manipulation by combining 5 simple functions from {dplyr}:

- `select()` to keep or discard columns
- `group_by()` to define groups of rows for the following verbs
- `filter()` to keep or discard rows
- `mutate()` to create new columns
- `summarise()` to compute summary statistics

Example:

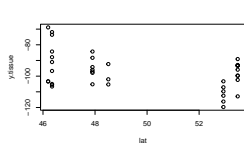
```
calibration |>
  filter(site != "A") |>
  mutate(y.tissue.not.permil = y.tissue / 1000) |>
  group_by(site) |>
  summarize(mean_per_site = mean(y.tissue.not.permil))
## # A tibble: 5 x 2
##   site mean_per_site
##   <chr>         <dbl>
## 1 B          -0.0999
## 2 C          -0.0972
## 3 D          -0.0947
## # ... with 2 more rows
```

16. How to plot things?

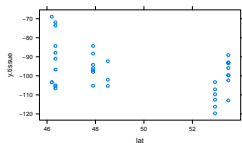
There are many plotting systems in **R**, such as:

- `{graphics}` (build-in system): most efficient for small jobs, but difficult for complex tasks
- `{lattice}`: difficult, but efficient for complex tasks
- `{ggplot2}`: easy and efficient for most tasks (but quite verbose)

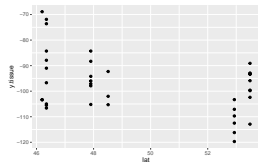
```
plot(y.tissue ~ lat,  
     data = calibration)
```



```
library(lattice)  
xyplot(y.tissue ~ lat,  
       data = calibration)
```



```
library(ggplot2)  
ggplot(data = calibration,  
       aes(x = lat, y = y.tissue)) +  
  geom_point()
```



Whatever you are using, plotting your data is very important! Defaults are always quite ugly!!

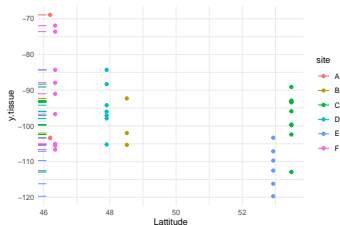
17. How to use {ggplot2} for an efficient production of plots?

All plots are composed of the same elements:

data + **aesthetic mappings** + **geometries** + **scales** + a **coordinate system** + a **theme** + a **faceting** specification
(the first 3 are mandatory, the last 4 are not since they have default presets)

We then build a plot by adding such components together:

```
ggplot(calibration) +  
  aes(x = lat, y = y.tissue, colour = site) +  
  geom_point() +  
  geom_rug(sides = "l") + ## l for left!  
  scale_x_continuous("Latitude") +  
  theme_minimal()
```



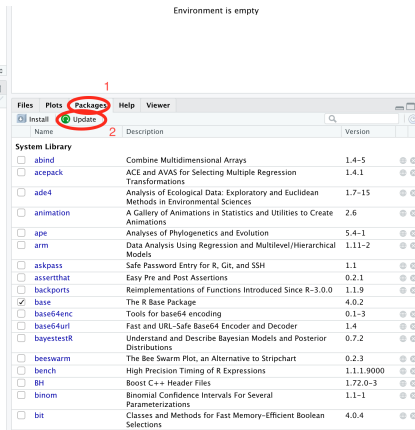
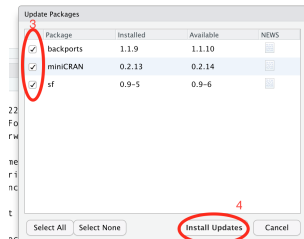
Note: check <https://www.r-graph-gallery.com> for examples

18. When to update what?

Update:

- R (at least once a year)
- RStudio (after each update of R)
- packages (at least once a month)

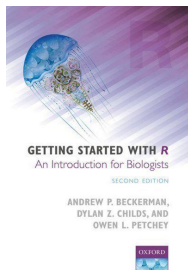
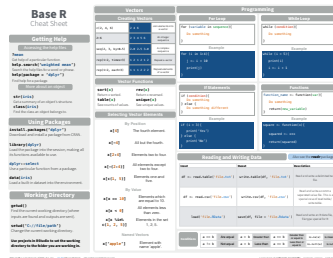
For packages:



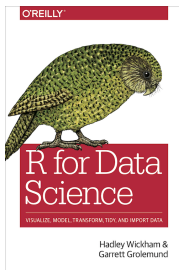
19. How to reach the level A2/B1?

There are many resources available at your disposal:

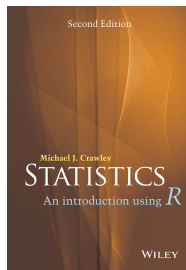
- tons of online tutorials, videos, forums. . .
- RStudio cheatsheets
- Official documentation (help files + <https://cran.r-project.org/manuals.html>)
- Books (not always free):



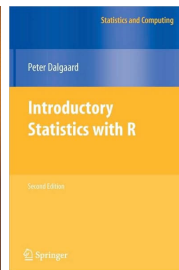
~ 30 €



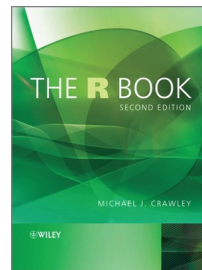
~ 0 or 60 €



~ 35 €



~ 40 €



~ 60 €

20. How to reach levels beyond B1?

This is not beyond your reach!

- try to help less advanced colleagues
- look at the code of more advanced **R** geeks
- act as a scientist: once you know some basics, if you notice something that does not behave as you thought it should, make hypotheses and test them
- try to contribute to collaborative **R** projects (e.g. on GitHub)

If you persist, you will become very good at **R** no matter how difficult you may find it now!