Introduction to **R**

Liam Bailey @rdataberlin

Leibniz Institute of Zoo and Wildlife Research

February 2021

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

Introduction to **R**

1. Why should you learn **R**?
2. Setting up **R** & RStudio
3. How best to organise your work
4. **R** basics
5. Using functions and package
6. Understanding a new function
7. Combining functions
8. How to import data into **R**
9. Manipulating vectors
10. Manipulating data frames
11. Manipulating lists
12. Plotting in R
13. **R** Markdown
14. When should I update?
15. How to reach level A2/B1
16. How to reach levels beyond B1

1. Why should you learn **R**?

- Best software out there for most data science tasks
- Open source and free
- Accessible without computer science background
- Rich in functionality with close to 100,000 free packages

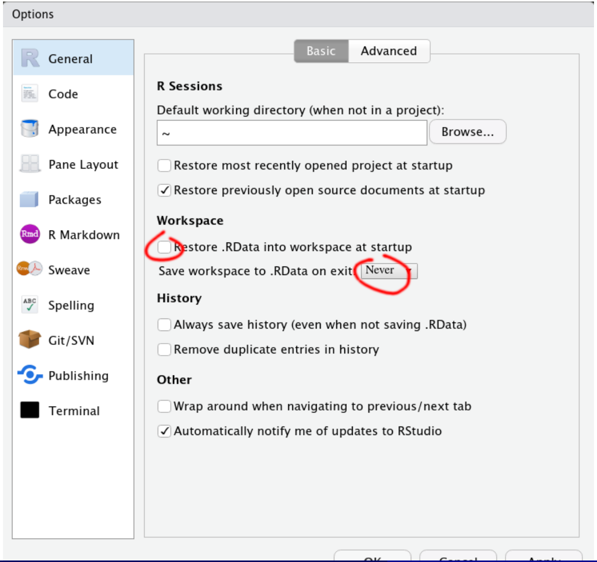
  (`https://rdrr.io`; `https://www.rdocumentation.org`)
- Friendly large community

  (twitter #rstats, `https://rfordatascience.slack.com`, `https://community.rstudio.com`, gatherings)

2. Setting up R & RStudio

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

Never save or restore the workspace!



R Sessions

Default working directory (when not in a project): ~

☐ Restore most recently opened project at startup

☑ Restore previously open source documents at startup

Workspace

☐ Restore .RData into workspace at startup

Save workspace to .RData on exit: Never

History

☐ Always save history (even when not saving .RData)

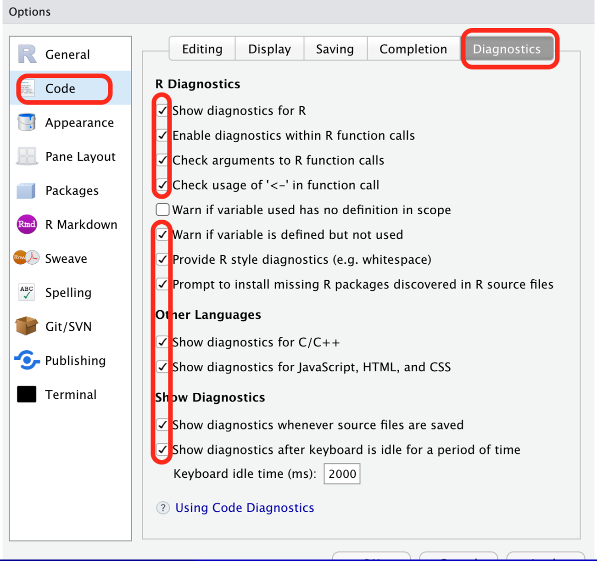
☐ Remove duplicate entries in history

Other

☐ Wrap around when navigating to previous/next tab

☑ Automatically notify me of updates to RStudio

Activate code diagnostics



R Diagnostics

☑ Show diagnostics for R

☑ Enable diagnostics within R function calls

☑ Check arguments to R function calls

☑ Check usage of '<-' in function call

☐ Warn if variable used has no definition in scope

☑ Warn if variable is defined but not used

☑ Provide R style diagnostics (e.g. whitespace)

☑ Prompt to install missing R packages discovered in R source files

Other Languages

☑ Show diagnostics for C/C++

☑ Show diagnostics for JavaScript, HTML, and CSS

Show Diagnostics

☑ Show diagnostics whenever source files are saved

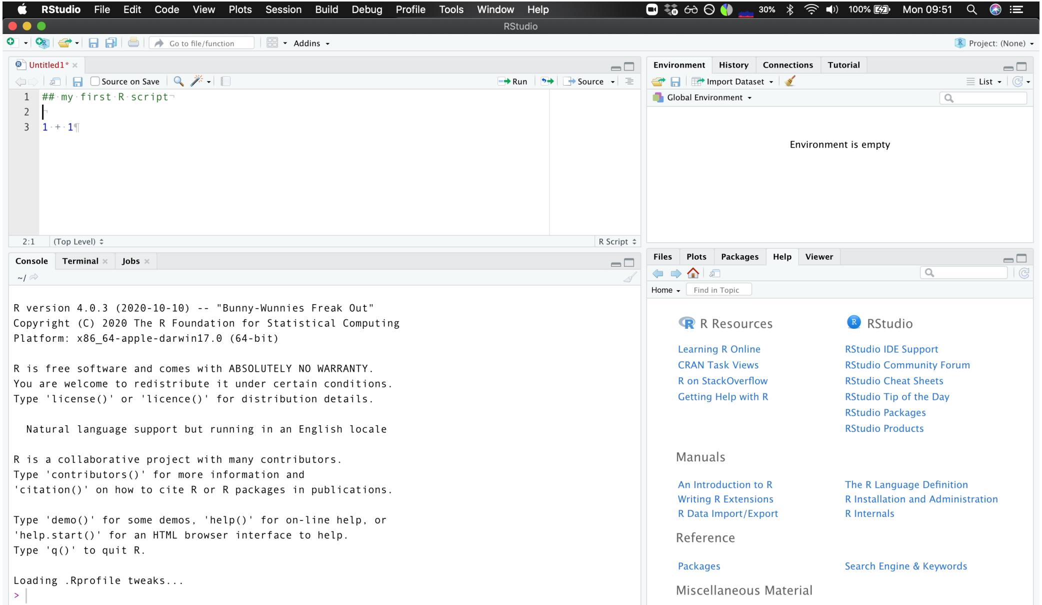
☑ Show diagnostics after keyboard is idle for a period of time

Keyboard idle time (ms): 2000

? Using Code Diagnostics

3. How best to organise your work

Use the RStudio Integrated Development Environment (IDE)



```r
## my first R script

1 + 1
```

```
R version 4.0.3 (2020-10-10) -- "Bunny-Wunnies Freak Out"
Copyright (C) 2020 The R Foundation for Statistical Computing
Platform: x86_64-apple-darwin17.0 (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.

  Natural language support but running in an English locale

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.

Loading .Rprofile tweaks...
>
```

3. How best to organise your work

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

1. Create a new RStudio project (Menu / File / New Project... / . . .)
2. Create a new **R** Script file (Menu / File / New File / R Script)
3. Save the created **R** Script file into the project folder directory

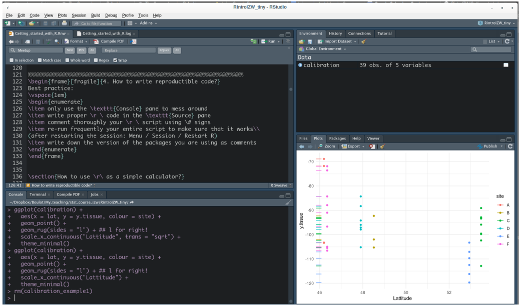
NB:

- An RStudio project is a folder containing the different files for a given project. Includes a project file that allows us to open RStudio with the correct working directory
- An **R** Script is a (text) file where we 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 <- more on this later!)

3. How best to organise your work

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


RintroIZW_tiny - RStudio
File Edit Code View Plots Session Build Debug Profile Tools Help
Go to file/function
Addins
RintroIZW_tiny
Getting_started_with_R.Rnw
Getting_started_with_R.log
Format
Compile PDF
Run
Meetup
Next Prev All
Replace
Replace All
□ In selection □ Match case □ Whole word □ Regex ☑ Wrap
126:41
How to write reproductible code?
R Sweave
Console Terminal Compile PDF Jobs
~/Dropbox/Boulot/My_teaching/stat_course_izw/RintroIZW_tiny/
Environment History Connections Tutorial
Import Dataset
List
Global Environment
Data
Files Plots Packages Help Viewer
Zoom Export
Publish
-70
-80
-90
-100
-110
-120
y.tissue
46
48
50
52
Lattitude
site
A
B
C
D
E
F

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
\begin{frame}[fragile]{4. How to write reproductible code?}
Best practice:
\vspace{1em}
\begin{enumerate}
\item only use the \texttt{Console} pane to mess around
\item write proper \r \ code in the \texttt{Source} pane
\item comment thoroughly your \r \ script using \# signs
\item re-run frequently your entire script to make sure that it works\\
(after restarting the session: Menu / Session / Restart R)
\item write down the version of the packages you are using as comments
\end{enumerate}
\end{frame}


\section{How to use \r\ as a simple calculator?}
```

```
calibration      39 obs. of 5 variables
```

```
> ggplot(calibration) +
+   aes(x = lat, y = y.tissue, colour = site) +
+   geom_point() +
+   geom_rug(sides = "l") + ## l for right!
+   scale_x_continuous("Lattitude", trans = "sqrt") +
+   theme_minimal()
> ggplot(calibration) +
+   aes(x = lat, y = y.tissue, colour = site) +
+   geom_point() +
+   geom_rug(sides = "l") + ## l for right!
+   scale_x_continuous("Lattitude") +
+   theme_minimal()
> rm(calibration_example1)
> 
```

4. **R** basics

R can perform basic arithmetic:

```
1 + 1
## [1] 2
1 - 1
## [1] 0
2 * pi
## [1] 6.28
3 / 2
## [1] 1.5
5^2
## [1] 25
5^(2 + 1)
## [1] 125
Inf/Inf
## [1] NaN
```

4. R basics

R can perform basic arithmetic:

```
1 + 1
## [1] 2
1 - 1
## [1] 0
2 * pi
## [1] 6.28
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
```

4. R basics

R can perform basic arithmetic:

```
1 + 1
## [1] 2
1 - 1
## [1] 0
2 * pi
## [1] 6.28
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
```

Practice

Compute $\sqrt{\frac{\frac{2^{3+1}}{4}}{5\times6} - 20}$

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

4. R basics

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

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

4. R basics

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:

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

4. **R** basics

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:

```
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
```

4. R basics

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

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

Objects can then be used through their name:

```r
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:

```r
(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

5. Using functions and package

For more complex tasks, we 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] 4.58
```

```
cv_x <- sd_x/mean_x
cv_x
## [1] 0.917
```

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

Note: all what is written in red above are functions

5. Using functions and package

Packages contain new functions! They extend the functionality of **R**

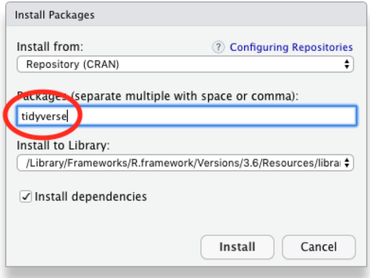
- For most users; e.g. `{dplyr}`, `{ggplot2}`
- For specific users; e.g. `{IsoriX}`, `{MixSIAR}`
- For developers creating packages; e.g. `{devtools}`, `{Rcpp}`

5. Using functions and package

Packages contain new functions! They extend the functionality of **R**

- 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):



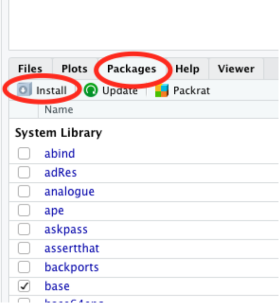
Install Packages

Install from: Repository (CRAN)

Packages (separate multiple with space or comma): tidyverse

Install to Library: /Library/Frameworks/R.framework/Versions/3.6/Resources/librar

☑ Install dependencies



System Library

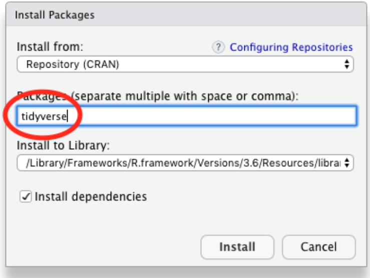
☐ abind
☐ adRes
☐ analogue
☐ ape
☐ askpass
☐ assertthat
☐ backports
☑ base

5. Using functions and package

Packages contain new functions! They extend the functionality of **R**

- 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):



Install Packages

Install from:

Repository (CRAN)

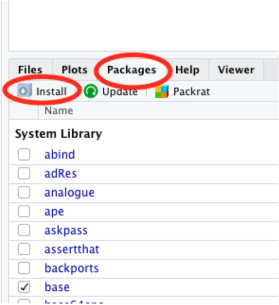
Packages (separate multiple with space or comma):

tidyverse

Install to Library:

/Library/Frameworks/R.framework/Versions/3.6/Resources/librai

☑ Install dependencies



System Library

- □ abind
- □ adRes
- □ analogue
- □ ape
- □ askpass
- □ assertthat
- □ backports
- ☑ base

Loading (once per session):

```r
library(dplyr)
```

Important packages for this course

- adegenet
- poppr
- pegas
- ggplot2
- lattice
- viridisLite

6. Understanding a new function

```r
mean()
```

```r
?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.
[...]
```

7. Combining functions

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
```

7. Combining functions

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:

```
library(magrittr) ## we load a package introducing pipes

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

7. Combining functions

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:

```
library(magrittr) ## we load a package introducing pipes

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 one line at a time

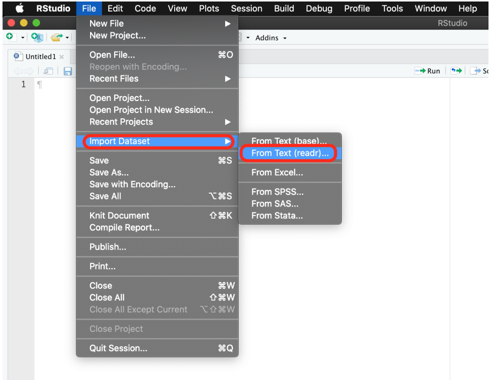
8. How to import data into R

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

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

For importing data you can often simply rely on the GUI

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



9. Manipulating vectors

A vector contains all elements of the same type:

```
## Elements will be coerced
my_vec <- c(1, 2, 3, "A", "B", "C")
my_vec
## [1] "1" "2" "3" "A" "B" "C"
```

9. Manipulating vectors

A vector contains all elements of the same type:

```
## Elements will be coerced
my_vec <- c(1, 2, 3, "A", "B", "C")
my_vec
## [1] "1" "2" "3" "A" "B" "C"
```

You can replace elements in a vector:

```
my_vec[1] <- NA ## change is permanent!
my_vec
## [1] NA  "2" "3" "A" "B" "C"
```

9. Manipulating vectors

A vector contains all elements of the same type:

```
## Elements will be coerced
my_vec <- c(1, 2, 3, "A", "B", "C")
my_vec
## [1] "1" "2" "3" "A" "B" "C"
```

You can replace elements in a vector:

```
my_vec[1] <- NA ## change is permanent!
my_vec
## [1] NA  "2" "3" "A" "B" "C"
```

You can select one or multiple elements:

```
my_vec[2]
## [1] "2"
my_vec[1:3] ## change is not saved unless you assign it
## [1] NA  "2" "3"
```

9. Manipulating vectors

A vector contains all elements of the same type:

```
## Elements will be coerced
my_vec <- c(1, 2, 3, "A", "B", "C")
my_vec
## [1] "1" "2" "3" "A" "B" "C"
```

You can replace elements in a vector:

```
my_vec[1] <- NA ## change is permanent!
my_vec
## [1] NA  "2" "3" "A" "B" "C"
```

You can select one or multiple elements:

```
my_vec[2]
## [1] "2"
my_vec[1:3] ## change is not saved unless you assign it
## [1] NA  "2" "3"
```

Select all except:

```
my_vec[-1]
## [1] "2" "3" "A" "B" "C"
```

9. Manipulating vectors

You can select multiple using a vector of `logicals`:

```
my_vec %in% c("A", "B", "C")
## [1] FALSE FALSE FALSE  TRUE  TRUE  TRUE
my_vec[my_vec %in% c("A", "B", "C")]
## [1] "A" "B" "C"
```

10. Manipulating data frames

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

10. Manipulating 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
```

10. Manipulating 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)

10. Manipulating data frames

We can manipulate data frames using inbuilt R functionality

```
## extract vector
calibration$y.tissue
##  [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
```

10. Manipulating data frames

We can manipulate data frames using inbuilt R functionality

```
## extract vector
calibration$y.tissue
##  [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
```

```
## filter rows and columns data frame
calibration[calibration$site == "B", c("lat", "long", "y.tissue")]
## # 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.
```

10. Manipulating data frames

We can manipulate data frames using inbuilt R functionality

```
## extract vector
calibration$y.tissue
##  [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
```

```
## filter rows and columns data frame
calibration[calibration$site == "B", c("lat", "long", "y.tissue")]
## # 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$sd.y.tissue <- NULL ## delete a column. Change is permanent!
calibration$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
```

10. Manipulating data frames

You can achieve all kinds 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

10. Manipulating data frames

You can achieve all kinds 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
```

11. Manipulating 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)
```

11. Manipulating lists

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

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

```r
## reveals the (often hidden) structure of the list
str(test_calib2sp)
## 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"
##  $ 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"
```

11. Manipulating 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)
```

```
## reveals the (often hidden) structure of the list
str(test_calib2sp)
## 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"
##  $ 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:

```
test_calib2sp$p.value
## [1] 0.36
```

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

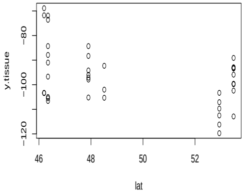
```
test_calib2sp[[3]]
## [1] 0.36
```

12. Plotting in R

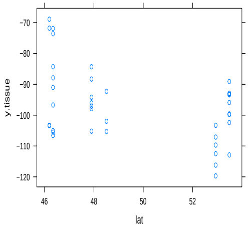
There are many plotting systems in **R**, such as:

- {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)

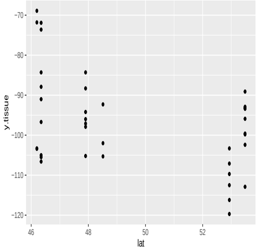
```r
plot(y.tissue ~ lat,
     data = calibration)
```



```r
library(lattice)

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



```r
library(ggplot2)

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



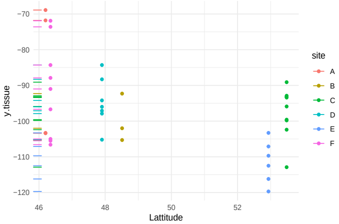
Whatever you are using, plotting your data in very important! Defaults are always quite ugly!!

12. Plotting in R

{ggplot2} is a powerful tool for plotting Idea: all plots are composed of the same elements

(data + **aes**thetic mappings + **scales** + a **coord**inate system + a **facet**ing specification + a **theme**)

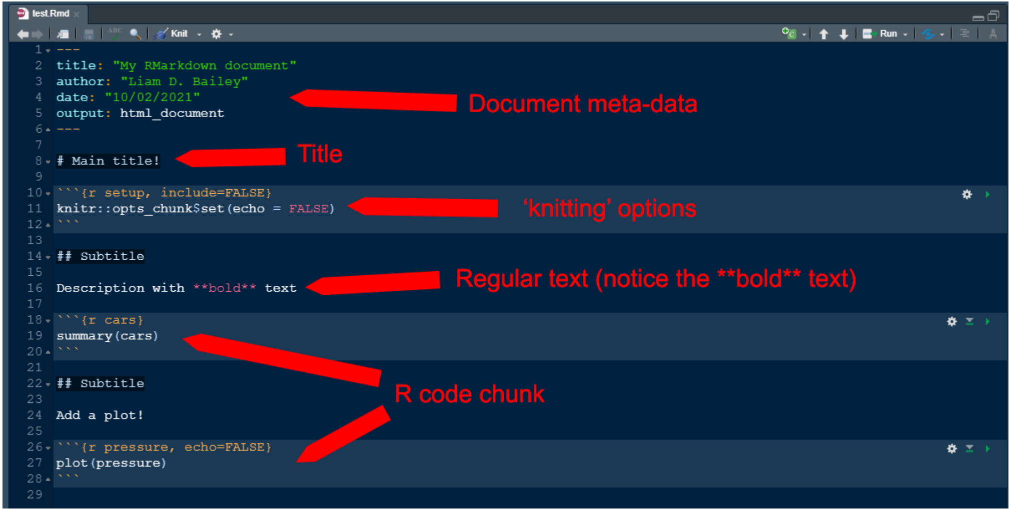
```r
ggplot(calibration) +
  aes(x = lat, y = y.tissue, colour = site) +
  geom_point() +
  geom_rug(sides = "l") + ## l for left!
  scale_x_continuous("Lattitude") +
  theme_minimal()
```



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

13. R Markdown

R Markdown is a powerful tool to combine code and text using Markdown language and generate HTML or PDF documents



~~~
---
title: "My RMarkdown document"
author: "Liam D. Bailey"
date: "10/02/2021"
output: html_document
---

# Main title!

```{r setup, include=FALSE}
knitr::opts_chunk$set(echo = FALSE)
~~~

Subtitle

Description with **bold** text

```{r cars}
summary(cars)
```

Subtitle

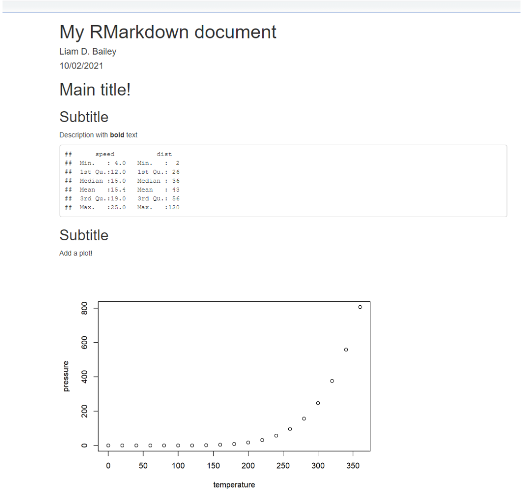
Add a plot!

```{r pressure, echo=FALSE}
plot(pressure)
```

```
```

13. **R** Markdown

R Markdown is a powerful tool to combine code and text using Markdown language and generate HTML or PDF documents



My RMarkdown document

Liam D. Bailey

10/02/2021

Main title!

Subtitle

Description with **bold** text

```
##      speed           dist       
##  Min.   : 4.0   Min.   :  2  
##  1st Qu.:12.0   1st Qu.: 26  
##  Median :15.0   Median : 36  
##  Mean   :15.4   Mean   : 43  
##  3rd Qu.:19.0   3rd Qu.: 56  
##  Max.   :25.0   Max.   :120  
```

Subtitle

Add a plot!

13. **R** Markdown

Markdown tricks:

- #title
- ##subtitle
- _italic_
- **bold**
- **_italic and bold_**

13. **R** Markdown

R chunk tricks for knitting:

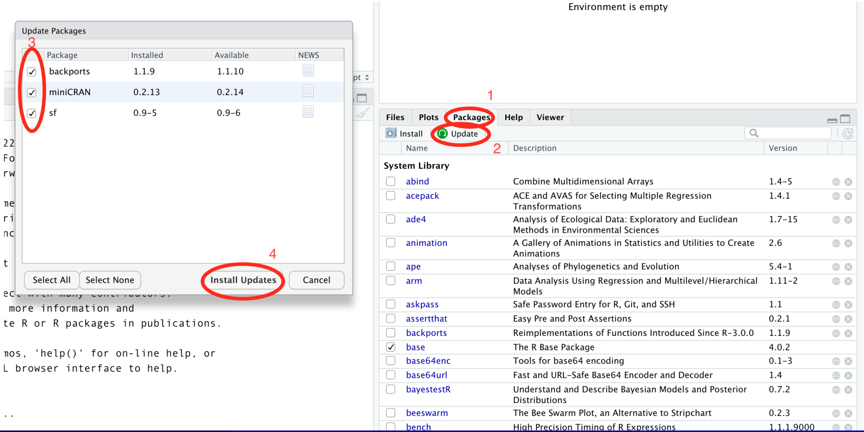
- *echo* = *FALSE*: Hide the code but show the output
- *eval* = *FALSE*: Don't run the code
- *include* = *FALSE*: Run the code but hide the code and output
- *fig.height*/*fig.width*: Adjust the size of figures

14. When should I update?

Update:

- **R** (at least once a year)
- RStudio (at least twice a year)
- packages (at least once a month)

For packages:



☐	Package	Installed	Available	NEWS
☑	backports	1.1.9	1.1.10	
☑	miniCRAN	0.2.13	0.2.14	
☑	sf	0.9-5	0.9-6	

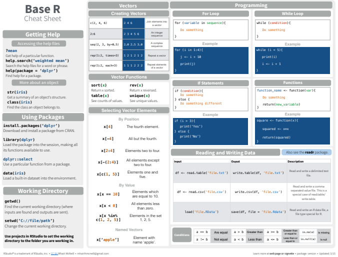
```
22
Fo
rw

me
ri
nc

t

ec
 more information and
te R or R packages in publications.

mos, 'help()' for on-line help, or
L browser interface to help.

..
```

	Name	Description	Version
☐	abind	Combine Multidimensional Arrays	1.4-5
☐	acepack	ACE and AVAS for Selecting Multiple Regression Transformations	1.4.1
☐	ade4	Analysis of Ecological Data: Exploratory and Euclidean Methods in Environmental Sciences	1.7-15
☐	animation	A Gallery of Animations in Statistics and Utilities to Create Animations	2.6
☐	ape	Analyses of Phylogenetics and Evolution	5.4-1
☐	arm	Data Analysis Using Regression and Multilevel/Hierarchical Models	1.11-2
☐	askpass	Safe Password Entry for R, Git, and SSH	1.1
☐	assertthat	Easy Pre and Post Assertions	0.2.1
☐	backports	Reimplementations of Functions Introduced Since R-3.0.0	1.1.9
☑	base	The R Base Package	4.0.2
☐	base64enc	Tools for base64 encoding	0.1-3
☐	base64url	Fast and URL-Safe Base64 Encoder and Decoder	1.4
☐	bayestestR	Understand and Describe Bayesian Models and Posterior Distributions	0.7.2
☐	beeswarm	The Bee Swarm Plot, an Alternative to Stripchart	0.2.3
☐	bench	High Precision Timing of R Expressions	1.1.1.9000

15. How to reach 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):



Base R
Cheat Sheet

Getting Help

Accessing the help files

`?mean`
Get help of a particular function.
`help.search('weighted mean')`
Search the help files for a word or phrase.
`help(package = 'dplyr')`
Find help for a package.

More about an object

`str(iris)`
Get a summary of an object's structure.
`class(iris)`
Find the class an object belongs to.

Using Packages

`install.packages('dplyr')`
Download and install a package from CRAN.

`library(dplyr)`
Load the package into the session, making all its functions available to use.

`dplyr::select`
Use a particular function from a package.

`data(iris)`
Load a built-in dataset into the environment.

Working Directory

`getwd()`
Find the current working directory (where inputs are found and outputs are sent).

`setwd('C://file/path')`
Change the current working directory.

Use projects in RStudio to set the working directory to the folder you are working in.

Vectors

Creating Vectors

`c(2, 4, 6)`	2 4 6	Join elements into a vector
`2:6`	2 3 4 5 6	An integer sequence
`seq(2, 3, by=0.5)`	2.0 2.5 3.0	A complex sequence
`rep(1:2, times=3)`	1 2 1 2 1 2	Repeat a vector
`rep(1:2, each=3)`	1 1 1 2 2 2	Repeat elements of a vector

Vector Functions

`sort(x)`
Return x sorted.
`rev(x)`
Return x reversed.
`table(x)`
See counts of values.
`unique(x)`
See unique values.

Selecting Vector Elements

By Position

`x[4]`	The fourth element.
`x[-4]`	All but the fourth.
`x[2:4]`	Elements two to four.
`x[-(2:4)]`	All elements except two to four.
`x[c(1, 5)]`	Elements one and five.

By Value

`x[x == 10]`	Elements which are equal to 10.
`x[x < 0]`	All elements less than zero.
`x[x %in% c(1, 2, 5)]`	Elements in the set 1, 2, 5.

Named Vectors

`x['apple']`	Element with name 'apple'.

Programming

For Loop

```
for (variable in sequence){

   Do something

}
```

Example

```
for (i in 1:4){

   j <- i + 10

   print(j)

}
```

While Loop

```
while (condition){

   Do something

}
```

Example

```
while (i < 5){

   print(i)

   i <- i + 1

}
```

If Statements

```
if (condition){
   Do something
} else {
   Do something different
}
```

Example

```
if (i > 3){
   print('Yes')
} else {
   print('No')
}
```

Functions

```
function_name <- function(var){

   Do something

   return(new_variable)
}
```

Example

```
square <- function(x){

   squared <- x*x

   return(squared)
}
```

Reading and Writing Data

Also see the **readr** package.

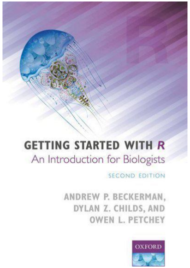
Input	Ouput	Description
`df <- read.table('file.txt')`	`write.table(df, 'file.txt')`	Read and write a delimited text file.
`df <- read.csv('file.csv')`	`write.csv(df, 'file.csv')`	Read and write a comma separated value file. This is a special case of read.table/ write.table.
`load('file.RData')`	`save(df, file = 'file.Rdata')`	Read and write an R data file, a file type special for R.

Conditions

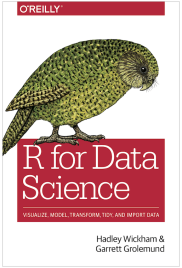
`a == b`	Are equal	`a > b`	Greater than	`a >= b`	Greater than or equal to	`is.na(a)`	Is missing
`a != b`	Not equal	`a < b`	Less than	`a <= b`	Less than or equal to	`is.null(a)`	Is null

RStudio® is a trademark of RStudio, Inc. • CC BY Mhairi McNeill • mhairihmcneill@gmail.com

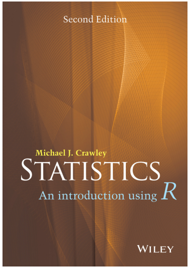
Learn more at **web page or vignette** • package version • Updated: 3/15



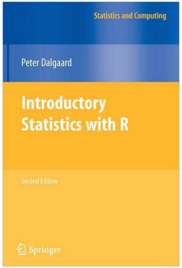
∼ 30 €



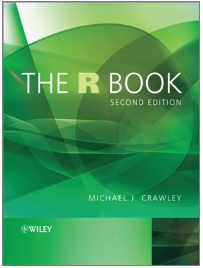
∼ 0 *or* 60 €



∼ 35 €



∼ 40 €



∼ 60 €

16. 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** not matter how difficult you may find it now!