R INTRODUCTION

Tobias Niedermaier

INTRO

WHY R?

Why R?

- R is **open source**
- All techniques for data analyses
- State-of-the-art graphics capabilities
- A platform for programming new statistical methods or analysis pipelines (in form of R-packages)

PROGRAMMING (IN GENERAL)

"Good programmers are made, not born." (Gerald M. Weinberg - The Psychology of Computer Programming)

- consequence I

  train…

- consequence II

  train…

- consequence III

  train more

Hands-on is important. Understanding is less that 30%

R AND R STUDIO

Required tools for the course:

- Programming language R
  - designed to make fast prototyping for statistical analysis
  - interpreted language
- RStudio (optional, but recommended)
  - IDE tailored for `R`
  - Integrates a lot more (e.g. python, c++, etc.)
- Alternatively, you may use any code editor that supports R and copy-paste your codes into the command line (Notepad++, Emacs etc.)

R PACKAGES

- `R` comes with many useful packages by default
- However, the strength lies in the huge collection of external packages
- Most popular and default: **CRAN**
- Install new packages in R using either
  - using a command:
    - `install.packages("<package-name>")` (e.g. `install.packages("mvtnorm")`)
  - RStudio
    - using built-in tools from the IDE

BASIC OPERATIONS

ADDITION, SUBTRACTION, ETC

```
1+2
```

```
[1] 3
```

```
1-2
```

```
[1] -1
```

```
1*2
```

```
[1] 2
```

```
1/2
```

```
[1] 0.5
```

```
1^2
```

```
[1] 1
```

```
1**2
```

```
[1] 1
```

> **Note**
>
> What will happen?
>
> ```
> 1/0
> ```

SPECIAL SYMBOLS FUNCTIONS

Special symbols

```r
pi
Inf
```

Mathematical functions

```r
exp(1)
```

```
[1] 2.718282
```

```r
log(1)
```

```
[1] 0
```

Special cases:

- `NaN` is a data type that indicates an invalid number.

```r
log(-1)
```

```
[1] NaN
```

```r
NaN + 1
```

```
[1] NaN
```

- `NA` is a missing value.

```r
NA + 1
```

```
[1] NA
```

- `NULL` means literally empty/nothing

ASSIGNING OBJECTS

Assignment is done using `<-`

```r
x <- 1
y <- 2
x + y
```

```
[1] 3
```

Alternatively, use `=`

```r
x = sqrt(2)
y = sqrt(2)
x * y
```

```
[1] 2
```

Or (almost never used):

```r
5 -> x
```

Look at environment pane in R Studio, what can you see?

Also try:

```r
ls()
```

```
[1] "x" "y"
```

```r
ls.str()
```

```
x :  num 5
y :  num 1.41
```

NAMING OBJECTS

- Objects in R have to start with a letter

Case sensitive

```
a <- 2
A <- 1
a-A
```

```
[1] 1
```

Overwrite variables with old ones

```
a <- a + 1 #There is NO warning if you overwrite an existing object!
```

Combination of words

```
variable_name <- 1
variable.name <- 1
variableName <- 1
```

COMMENTS

Sometimes it is useful, to comment code. Use a `#` to comment

Standard:

```
1+1
```

```
[1] 2
```

Comment a line (no output):

```
# 1+1
```

Comment after an expression (only `1+1` gets evaluated):

```
1+1 # +1
```

```
[1] 2
```

FUNCTION CALLING

So far we used expressions like `f(...)`. This is a **function**. E.g.

```r
exp(2)
```

We call the function `exp` with a value of 2. Or the (natural) logarithm:

```r
log(exp(1))
```

```
[1] 1
```

We can specify the base as a *second argument*:

```r
log(2, 2)
```

```
[1] 1
```

> **Note**
>
> What will happen?
>
> ```r
> Log(Exp(1))
> ```

GET DOCUMENTATION

Access the documentation using

- `<F1>`
- type `?function_name`
- use RStudio functionality

E.g. documentation for `log()` reveals that we calculate the natural logarithm.

```r
?log
log(x, base = exp(1))
```

FUNCTION CALLING CONT'D

You can ignore the argument name, when placements are clear. - We have done that for `exp` and `log`

Hence, this here

```r
log(2, 2)
```

means, that we actually call

```r
log(x=2, base=2)
```

If you specify the argument, order does not matter.

Example:

```r
log(base=3, x=2)
log(3, 2)
```

Note

What will happen?

```r
log <- 1
log(log)
```

BASIC (PRIMITIVE) DATA TYPES

`numeric`

A (floating point) number. We used this so far (default).

`1.0, 1.34, -33, pi`

`logical`

A binary data type.

`TRUE, FALSE, T, F`

> **Note**
>
> What will happen?
>
> ```
> sum(c(TRUE, FALSE, FALSE, TRUE, FALSE))
> sum(!c(TRUE, FALSE, FALSE, TRUE, FALSE))
> ```

`integer`

Can be specified using an "L".

`1L, 100L, -99L`

`character`

Represents letters OR sentences.

`'a', "abc", "May the force be with you"`

EXERCISES 1 TASK 1

VECTORS

VECTORS

You can *combine* single values to a *vector.*

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

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

```
b <- c(TRUE, FALSE, TRUE)
b
```

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

```
c <- c("a", 'ab', "ab c")
c
```

```
[1] "a"    "ab"   "ab c"
```

Many operations in R are *vectorized*

```
a + a
```

```
[1] 2 4 6 8
```

```
a * a
```

```
[1]  1  4  9 16
```

```
exp(a)
```

```
[1]  2.718282  7.389056 20.085537 54.598150
```

```
-a
```

```
[1] -1 -2 -3 -4
```

Note

What will happen?

```
c("1",2,3)
```

VECTORS CONT'D

- NA or NaNs can be part of a vector

```
a <- c(1,2,NA,4)
a + 1
```

```
[1]  2  3 NA  5
```

```
b <- c(1, -1, Inf)
log(b)
```

```
[1]   0 NaN Inf
```

AUTOMATIC RECYCLING

```
a <- c(1,2,3,4)
a + 1
```

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

```
b <- c(2,2)
a + b
```

```
[1] 3 4 5 6
```

Warning

Note the behavior for vectors with different length! Example:

```
a <- c(1,2,3)
b <- c(1,2)
a + b
```

```
Warning in a + b: Länge des längeren Objektes
     ist kein Vielfaches der Länge des kürzeren Objektes
```

```
[1] 2 4 4
```

VECTOR CREATION

There are a lot of convenience functions to create vectors.

```
c(1,2,3,4)
```

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

```
1:4
```

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

```
seq(4)
```

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

More complex ones:

```
4:-3
```

```
[1]  4  3  2  1  0 -1 -2 -3
```

```
seq(-10, 10, by = 2)
```

```
 [1] -10  -8  -6  -4  -2   0   2   4   6   8  10
```

```
seq(-10, 10, length.out = 10) # vector of length 10
```

```
 [1] -10.000000  -7.777778  -5.555556  -3.333333  -1.111111   1.111111
 [7]   3.333333   5.555556   7.777778  10.000000
```

SELECT ELEMENTS OF A VECTOR

Access elements of a vector using positional numbers within `[...]`:

```
x <- c(2,4,2,5)
x[1]
```

```
[1] 2
```

Multiple elements

```
selection <- c(1,4)
x[selection]
```

```
[1] 2 5
```

```
x[c(1,4)]
```

```
[1] 2 5
```

Negative values will be excluded

```
x[-c(1,3)]
```

```
[1] 4 5
```

> **Note**
>
> What will happen?

```
x[1:5]
x[-(5:10)]
x[0]
```

LOGICAL VALUES FOR COMPARISON

Recall the very most basic data type `logical`, i.e. `TRUE` and `FALSE`.

- We can create such an object by comparison:

```
1 == 2  # lhs equal rhs?
```

```
[1] FALSE
```

```
1 != 2  # lhs unequal rhs?
```

```
[1] TRUE
```

```
1 > 2  # lhs larger rhs?
```

```
[1] FALSE
```

```
1 >= 2  # lhs larger or equal rhs?
```

```
[1] FALSE
```

```
1 < 2  # lhs less than rhs?
```

```
[1] TRUE
```

```
1 <= 2  # lhs less or equal than rhs?
```

```
[1] TRUE
```

Swap value:

```r
!TRUE
```

```
[1] FALSE
```

```r
!FALSE
```

```
[1] TRUE
```

Note

What will happen?

```r
1 == "1"
1 != NaN
NA == NA  # we will learn the solution in a few slides
```

FILTER ELEMENTS OF A VECTOR

Comparison operators are vectorized:

```
c(T,F,T) == c(F,F,T)  # element-wise comparison
```

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

Except if you use identical() which is FALSE if there is any mismatch:

```
identical(c(T,F,T), c(F,F,T))
```

```
[1] FALSE
```

Check condition on a numeric vector

```
x <- c(2,4,2,5)
position_two <- x == 2  # logical vector showing, where the condition holds
position_two
```

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

Use logical values to filter a vector.

```
x[position_two]
```

```
[1] 2 2
```

```
# or directly
x[x == 2]
```

```
[1] 2 2
```

Filter for values less than 3

```r
x[x < 3]
```

```
[1] 2 2
```

COMBINE FILTERS WITH & AND |

Combination operations…

```r
TRUE & TRUE
```

```
[1] TRUE
```

```r
FALSE & TRUE
```

```
[1] FALSE
```

```r
TRUE | TRUE
```

```
[1] TRUE
```

```r
FALSE | TRUE
```

```
[1] TRUE
```

…or vectorized

```r
x <- c(T,F,T,F)
y <- c(T,T,F,F)
x & y
```

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

```r
x | y
```

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

Use this to filter a vector for multiple conditions

```r
x[(x < 5) & (x > 2)]
```

```
logical(0)
```

ASSIGN NEW VALUES IN A VECTOR

We can assign new values to a vector using a combination of selection and assignment

```
x <- 1:5
x[1] <- 2
x
```

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

```
x[x > 3] <- -99
x
```

```
[1]   2   2   3 -99 -99
```

```
x[-1] <- 100
x
```

```
[1]   2 100 100 100 100
```

> **Note**
>
> What will happen?
>
> ```
> x[100] <- 1
> ```

VECTOR OPERATIONS

```r
x <- c(1,1,2,3)
length(x)
```

```
[1] 4
```

```r
append(x, c(1,2,3))
```

```
[1] 1 1 2 3 1 2 3
```

```r
rev(x)
```

```
[1] 3 2 1 1
```

```r
sort(x)
```

```
[1] 1 1 2 3
```

```r
unique(x)
```

```
[1] 1 2 3
```

```r
sum(x)
```

```
[1] 7
```

EXERCISES 1 TASK 2

COMPLEX STRUCTURES

FACTORS

Consider a vector, that represents a categorical variable. Let's say colors.

```r
colors <- c("blue", "red", "blue", "red", "green", "black", "green", "white")
colors
```

```
[1] "blue"  "red"   "blue"  "red"   "green" "black" "green" "white"
```

We cast `colors` into a factor now:

```r
colors <- as.factor(colors)
colors
```

```
[1] blue  red   blue  red   green black green white
Levels: black blue green red white
```

```r
levels(colors)
```

```
[1] "black" "blue"  "green" "red"   "white"
```

```r
as.numeric(colors)
```

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

```r
class(colors)
```

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

```r
typeof(colors)
```

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


Hence, a vector of integeres where each value corresponds to a

COMPLEX DATA STRUCTURES

from *Ceballos and Cardiel, (2013). Data structure – First Steps in R. Retreived 25-11-2018 from http://venus.ifca.unican.es/Rintro/_images/dataStructuresNew.png*

Use `str(...)` to inspect the structure of complex data types!

VECTOR, MATRIX, ARRAY

We already got vectors. Lets combine them:

```
x <- 1:4
(x_rbind <- cbind(x,x)) # 4 rows, 2 columns
```

```
     x x
[1,] 1 1
[2,] 2 2
[3,] 3 3
[4,] 4 4
```

```
(x_cbind <- rbind(x,x)) # 2 rows, 4 columns
```

```
  [,1] [,2] [,3] [,4]
x    1    2    3    4
x    1    2    3    4
```

```
dim(x_rbind)
```

```
[1] 4 2
```

```
dim(x_cbind)
```

```
[1] 2 4
```

```
nrow(x_rbind)
```

```
[1] 4
```

```
ncol(x_rbind)
```

```
[1] 2
```

VECTOR, MATRIX, ARRAY CONT'D

We can define a matrix using the `matrix` function:

```
matrix(1:6, nrow = 3, ncol = 2)
```

```
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
```

```
matrix(1:6, nrow = 3, ncol = 2, byrow = T)
```

```
     [,1] [,2]
[1,]    1    2
[2,]    3    4
[3,]    5    6
```

Arrays as a generalization with multiple dimensions

```
array(1:12, dim = c(3,2,2))
```

```
, , 1

     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6

, , 2
```

```
     [,1] [,2]
[1,]    7   10
[2,]    8   11
[3,]    9   12
```

This is also sometimes called a *tensor*.

SELECT/FILTER ELEMENTS ON ARRAYS

As vectors, we can select and filter. Seperate dimensions with a `,`, i.e. `[... , ...]`

```
(m <- matrix(1:6, nrow = 3, ncol = 2))
```

```
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
```

```
m[2,2]
```

```
[1] 5
```

```
m[nrow(m), ncol(m) ]
```

```
[1] 6
```

Defining no entry will return the full dimension:

```
m[2,]
```

```
[1] 2 5
```

```
m[,1]
```

```
[1] 1 2 3
```

Note

What will happen?

```
m[1,,2]
m[10]
```

LIST

A list is a collection of elements. These elements could be any object.

```
(l <- list(1, "2", 1:3, list(m)))
```

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

[[2]]
[1] "2"

[[3]]
[1] 1 2 3

[[4]]
[[4]][[1]]
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
```

Access elements of a list with `[[...]]`.

```
l[[2]]
```

```
[1] "2"
```

A sub-list can be accessed with `[...]`.

```r
l[1:3]
```

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

[[2]]
[1] "2"

[[3]]
[1] 1 2 3
```

LIST CONT'D

You can define names for lists:

```
l <- list(slot1 = 1:3, slot2 = c("a", "b"), slot3 = l)
names(l)
```

```
[1] "slot1" "slot2" "slot3"
```

Access list elements using the name and a `$`:

```
l$slot3 # return the original list l before overwriting it
```

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

[[2]]
[1] "2"

[[3]]
[1] 1 2 3

[[4]]
[[4]][[1]]
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
```

Delete elements by assigning a `NULL` to a slot

```
l[2:3] <- NULL
l
```

```
$slot1
[1] 1 2 3
```

DATA FRAME

A data frame is basically a list, where each element is a vector of the same length. However, it implements function to handle it as a matrix.

Let's define a data set representing cars:

```
col <- as.factor(c("blue", "red", "blue", "red", "green", "black", "green", "white"))
pri <- c(10, 20, 9, 50, 0.4, 15, 160, 60) * 1000
is_el <- c(F,F,F,T,F,T,F,T)

car_ds <- data.frame(color = col, price = pri, is_electric = is_el)
car_ds
```

```
  color  price is_electric
1  blue  10000       FALSE
2   red  20000       FALSE
3  blue   9000       FALSE
4   red  50000        TRUE
5 green    400       FALSE
6 black  15000        TRUE
7 green 160000       FALSE
8 white  60000        TRUE
```

```
str(car_ds)
```

```
'data.frame':	8 obs. of  3 variables:
 $ color      : Factor w/ 5 levels "black","blue",..: 2 4 2 4 3 1 3 5
 $ price      : num  10000 20000 9000 50000 400 15000 160000 60000
 $ is_electric: logi  FALSE FALSE FALSE TRUE FALSE TRUE
```

DATA FRAME CONT'D

We can work on a data set as we work with a matrix

```
# All rows with red cars
car_ds[car_ds$color == "red", ]
```

```
  color price is_electric
2   red 20000       FALSE
4   red 50000        TRUE
```

```
# price of all black cars
car_ds[car_ds$color == "black", "price"]
```

```
[1] 15000
```

```
# set a new price for the last car in the ds
car_ds[8, 2] <- 600
car_ds
```

```
  color  price is_electric
1  blue  10000       FALSE
2   red  20000       FALSE
3  blue   9000       FALSE
4   red  50000        TRUE
5 green    400       FALSE
6 black  15000        TRUE
7 green 160000       FALSE
8 white    600        TRUE
```

MORE ON DATA STRUCTURES

- A data frame behaves like a matrix.
- However, keep in mind that it is actually a list. We can easily prove that:

```r
is.list(car_ds)
```

```
[1] TRUE
```

Use `str(...)` to check the data structure of *any* object:

```r
str(car_ds)
```

```
'data.frame':	8 obs. of  3 variables:
 $ color      : Factor w/ 5 levels "black","blue",..: 2 4 2 4 3 1 3 5
 $ price      : num  10000 20000 9000 50000 400 15000 160000 600
 $ is_electric: logi  FALSE FALSE FALSE TRUE FALSE TRUE ...
```

```r
m <- matrix(1:4, ncol = 2)
str(m)
```

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

You will frequently need conditional subsetting, combining several conditions:

```r
mtcars[mtcars$mpg > 15 & mtcars$wt < 3 & mtcars$hp <= 66,]
```

```
                mpg cyl disp hp drat    wt  qsec vs am gear carb
Fiat 128       32.4   4 78.7 66 4.08 2.200 19.47  1  1    4    1
Honda Civic    30.4   4 75.7 52 4.93 1.615 18.52  1  1    4    2
Toyota Corolla 33.9   4 71.1 65 4.22 1.835 19.90  1  1    4    1
Fiat X1-9      27.3   4 79.0 66 4.08 1.935 18.90  1  1    4    1
```

LOAD DATA

We can load a data set from a package using `data(...)`.

```
data("iris", package = "datasets")  # look in the environment variables
```

We can load data from files. Use `read.table(...)`, or wrapper functions with reasonable default values. E.g. We can read a file directly from the web:

```
d <- read.csv("https://raw.githubusercontent.com/vincentarelbundock/Rdatasets/master/csv/datasets/m
head(d)  # show the first few lines of a data set
```

```
           rownames  mpg cyl disp  hp drat    wt  qsec vs am gear carb
1         Mazda RX4 21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
2     Mazda RX4 Wag 21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
3        Datsun 710 22.8   4  108  93 3.85 2.320 18.61  1  1    4    1
4    Hornet 4 Drive 21.4   6  258 110 3.08 3.215 19.44  1  0    3    1
5 Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2
6           Valiant 18.1   6  225 105 2.76 3.460 20.22  1  0    3    1
```

Note, that we can also use this to read a data set from a local directory! To do that we have to specify either the full path or define the path from the **working directory**. Use `getwd(...)` and `setwd(...)` to get or set the current working directory. See next slide for an example.

SAVE DATA SETS

Consider a data set, you have worked with. You can save it using write functions.

```
write.csv(car_ds, file = "example_data.csv")  # we save our data set in the current working directo
```

We can again read the data as a new object:

```
d_loaded <- read.csv("example_data.csv")

all.equal(car_ds,d_loaded)  # test whether 2 (more complex) R object are the same
```

```
[1] "Names: 3 string mismatches"
[2] "Length mismatch: comparison on first 3 components"
[3] "Component 1: 'current' is not a factor"
[4] "Component 2: Modes: numeric, character"
[5] "Component 2: target is numeric, current is character"
[6] "Component 3: Modes: logical, numeric"
[7] "Component 3: target is logical, current is numeric"
```

We can read other files as well. E.g. excel, SPSS, SAS, etc.

There are a lot of packages to do that.

I use the function `load(...)` from the `rio` package that tries to unify a lot of different formats.)

SAVE AND LOAD R OBJECTS

So far, we only worked with data frames for read and write operations. We can save general `R` objects using `save(...)` and `load(...)` using the `.RData` format.

```r
a_list <- list(a = 42, data = iris, comment = "whatever")

save(a_list, file = "example_object.RData")

load("example_object.RData")
```

EXERCISES 1 TASKS 3