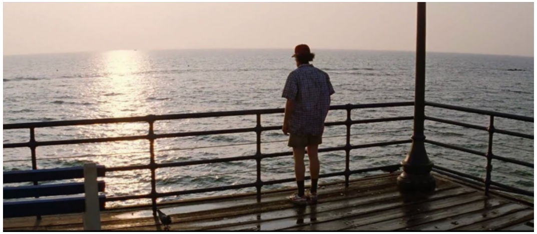
A GENTLE INTRODUCTION INTO THE STATISTICAL PROGRAMMING LANGUAGE R
LECTURE: UNSUPERVISED LEARNING AND EVOLUTIONARY COMPUTATION USING R

Jakob Bossek

MALEO Group, Department of Computer Science, Paderborn University, Germany

26th Oct, 2024

Why are there so many slides?



Forest reaching the ocean (for the first time ☺); Forest Gump, ©Paramount pictures 1994.

"That day, for no particular reason, I decided to go for a little run. So I ran to the end of the road. And when I got there, I thought maybe I'd run to the end of town. And when I got there, I thought maybe I'd just run across Greenbow County. And I figured, since I run this far, maybe I'd just run across the great state of Alabama. And that's what I did. I ran clear across Alabama. For no particular reason I just kept on going. I ran clear to the ocean. And when I got there, I figured, since I'd gone this far, I might as well turn around, just keep on going. When I got to another ocean, I figured, since I'd gone this far, I might as well just turn back, keep right on going." - **Forrest Gump**

Table of Contents

Introduction

Preface

- My original goal: at most 30 slides to teach the very essentials
- However, presentation slides evolved to a deck of $>$ 100 slides
- Consequence: we will skip a lot of stuff ☺
- Benefit for you: quite a lot important aspects are covered. Take it as a reference
- Many exercises without solutions (discuss with fellow students).[1]
- Enjoy! ☺

[1] 2nd exercise sheet will also contain many R exercises.

Introduction

- R is a statistical programming language[2] / environment[3]
- Includes a plethora of statistical features and graphical tools
- Interpreted language: offers command line interpreter (no compilation necessary)
- Nevertheless often quite fast (partially implemented in C and Fortran)
- Easy to extend via packages.[4]
- Open Source under GNU GPL v2

[2] R is no general purpuse language!

[3] According to R-manual: “”environment” is intended to characterize it as a fully planned and coherent system, rather than an incremental accretion of very specific and inflexible tools, as is frequently the case with other data analysis software.”

[4] Though R does a bad job at providing nice supporting tools. However, the community does via many useful packages.

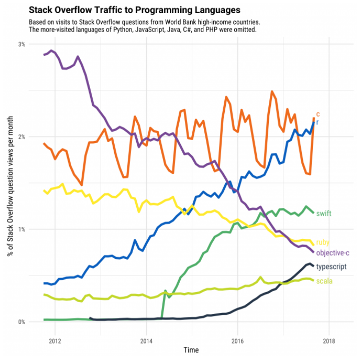
C'mon! Yet another programming language?

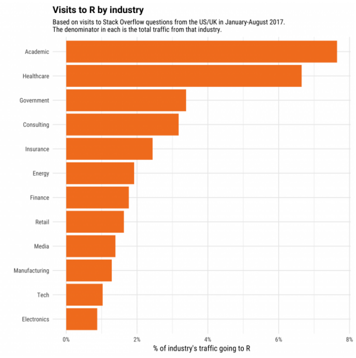
Why should I learn R? 2.0

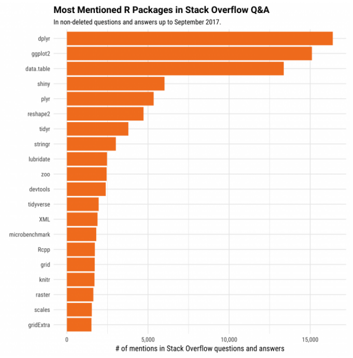
- Since August 2021, ranked 14th in the TIOBE programming language popularity index.[5]
- Increasing popularity in industry
- Easy to learn, mostly fun to use the language (in particular for users with few or no programming experience)
- Few lines of code can actually realize a lot (see next slide)
- Always a good idea to know multiple programming language.[6]
- We will mostly apply existing implementations: no need for deep algorithmics (only if you want to; see advanced exercises in lecture notes ☺)

The Impressive Growth of R (by Stack Overflow)

Study performed by Stack Overflow back in 2017 shows impressive growth of R:







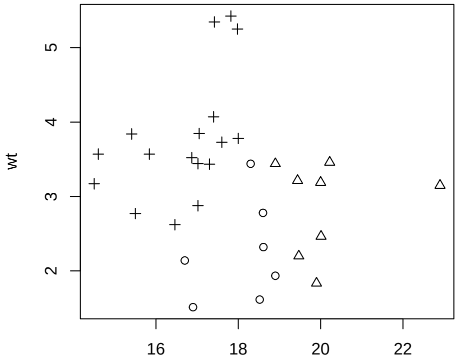
Figures taken from Stack Overflow blog post[7].

[7] https://stackoverflow.blog/2017/10/10/impressive-growth-r/

First impression

Achieve a lot with few lines of code:

```
> data(mtcars)
> X = mtcars[, c("qsec", "wt")]
> cl = kmeans(X, centers = 3, algorithm = "Lloyd")
> plot(X, pch = cl$cluster)
```



Editors with R support

The interactive shell is nice, but useless for larger projects. We want an editor with ...

- ... nice features: syntax-highlighting, code-completion etc.,
- possibility to save code in files and
- organize/maintain our data analysis project(s)

Nice editors

R-Studio powerful Integrated Development Environment (IDE) for R.

Sublime-Text Not an IDE! Very reduced, but minimalistic and lightning fast.

Visual Studio Code by Mircosoft; really good IDE with OKish R support.

Installation

R does not ship with the common operating systems

Windows Download latest pre-compiled binary version from CRAN

macOS Download latest pre-compiled binary version or use Homebrew[8]:

```
brew upgrade
brew install R
```

Linux Depends on the flavour of you distribution, but your OS package manager should help. E.g., on Ubuntu

```
sudo apt update
sudo apt install r-base
```

Some exotic OS I have no idea ☺

[8] You might want to use Homebrew if you do not yet ☺

Getting started: interacting with R

Start the interpreter by calling R from the command line[9] opens an interactive R session where you can type anything you want. At the *prompt* (>) we can perform calculations:

```
> 1 + 2 # this is a comment
## [1] 3

> 1 + 2 * 3 # R has operator precedence
## [1] 7

> # 1 + (2 * 3) # However, I like to make it explicit
> (1 + 2) * 3
## [1] 9

> log(10000) # function are called by passing arguments
## [1] 9.21034

> log(10000, base = 2)
## [1] 13.28771

> sqrt(25) # square root
## [1] 5

> 5^2 # power
## [1] 25
```

Looking for help

- Call `help.start()` to enter the documentation entry point.[10]
- For a function, say `mean`, you can view the documentation via `?mean`.[11]
- Exact function name unknown: use `??mean`
- Looking for something similar via `apropos(...)`, e.g.,

```
> apropos("var")
##  [1] "all.vars"         "combine_vars"     "estVar"           "get_all_vars"
##  [5] "globalVariables" "var"              "var.test"         "variable.names"
##  [9] "varimax"          "vars"
```

- Struggling with error message(s)?
  $\rightsquigarrow$ copy & paste into your favourite online search engine
  (might help to add "r" to your search request as an additional keyword)
- Ask ChatGPT

[10] Unfortunately, the help page layout is from the past century.

[11] Often documentations are full of details and might be overwhelming on first sight

Data types and classes

Atomic data types / vector types

As in all other programming languages there are several data types:

`numeric`	Real-valued numbers.
`integer`	Integer numbers.
`logical`	Boolean / logical values: `TRUE` and `FALSE`.
`character`	Strings / concatenation of letters.
`factor`	Special type of character which is internally stored as integer (for efficiency reasons); used for categorical variables.
`complex`	Complex numbers.
`raw`	Not discussed here.

Vectors: first look

Vectors (remember the mathematical definition?) are elementary building blocks in R. The very basic function to create a vector is c for combine / concatenate:

```
> c(14.4, 18.4, 19.5, 25.4) # numeric
## [1] 14.4 18.4 19.5 25.4

> c(1, 2, 5, 6) # actually also numeric
## [1] 1 2 5 6

> c(1L, 2L, 5L, 6L) # integer
## [1] 1 2 5 6

> c("dog", "cat", "rat", "mouse") # character
## [1] "dog"   "cat"   "rat"   "mouse"

> c(TRUE, FALSE, TRUE, FALSE, FALSE) # logical
## [1]  TRUE FALSE  TRUE FALSE FALSE

> x = 1:10 # save vector in variable
> x
##  [1]  1  2  3  4  5  6  7  8  9 10

> x = c(x, 11) # append another element to
> x
##  [1]  1  2  3  4  5  6  7  8  9 10 11
```

Vectors: implicit coercion

All elements of a vectors are of the same type $\rightsquigarrow$ in R all elements are coerced to the most general atomic data type:

```
> x = c(1, 2, 5, 6, 10.5)
> x
## [1]  1.0  2.0  5.0  6.0 10.5

> class(x)
## [1] "numeric"

> x = c(TRUE, "a", 5.43, 1)
> x
## [1] "TRUE" "a"    "5.43" "1"

> class(x)
## [1] "character"

> x = c(FALSE, 1)
> x
## [1] 0 1

> class(x)
## [1] "numeric"
```

Vectors: explicit coercion

Sometimes it makes sense to explicitely convert/coerce a vector into another data format:

```
> x_char = c("19.7", "35.45", "29.34", "24.99", "23")
> class(x_char)
## [1] "character"

> x_num = as.numeric(x_char)
> x_num
## [1] 19.70 35.45 29.34 24.99 23.00

> class(x_num)
## [1] "numeric"

> x_int = as.integer(x_num) # information loss
> x_int
## [1] 19 35 29 24 23

> class(x)
## [1] "numeric"
```

Vectors: sequencing

Generate sequences of numbers:

```
> seq(0, 1, by = 0.1)
##  [1] 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0

> seq(1, 0, by = -0.2)
## [1] 1.0 0.8 0.6 0.4 0.2 0.0

> seq(1, 5, length.out = 10)
##  [1] 1.000000 1.444444 1.888889 2.333333 2.777778 3.222222 3.666667 4.111111
##  [9] 4.555556 5.000000
```

Vectors: repetitions

`rep(...)` for repetition is handy to create special types of vectors:

```
> rep(4, 10)
##  [1] 4 4 4 4 4 4 4 4 4 4

> rep(4:6, 1:3)
## [1] 4 5 5 6 6 6

> rep(4:6, 3)
## [1] 4 5 6 4 5 6 4 5 6

> rep(4:6, each = 3)
## [1] 4 4 4 5 5 5 6 6 6

> # Use c to concatenate vectors
> x = rep(4, 3)
> c(10, x, rep(2:3, each =3))
##  [1] 10  4  4  4  2  2  2  3  3  3
```

Working with vectors

R offers many predefined functions to work with vectors:

```
> x = rep(c(1, 2), each = 4); y = 1:8
> z = x + y; z # vector  addition
## [1]  2  3  4  5  7  8  9 10

> 2 * x # scalar multiplication
## [1] 2 2 2 2 4 4 4 4

> x^2 # same as x * x
## [1] 1 1 1 1 4 4 4 4

> c(2, 3) * x # the shorter vector is repeated
## [1] 2 3 2 3 4 6 4 6

> sum(x) # sum :)
## [1] 12

> mean(x) # arithmetic mean
## [1] 1.5

> var(x) # sample variance
## [1] 0.2857143
```

Numeric functions

Function	**Description**
`abs(x)`	Absolute value
`sqrt(x)`	Square root
`log(x)`	Natural logarithm (base e)
`log10(x)`	Logarithm with base 10
`exp(x)`	Exponential funciton e^x
`cos(x), sin(x), ...`	Trigonometric functions
`ceiling(x)`	Round up: `ceiling(6.475)` is 7
`floor(x)`	ound down: `floor(6.489)` is 6
`trunc(x)`	Cut decimals: `trunc(2.99)` is 2
`round(x, digits=n)`	Regular rounding: `round(7.657, 2)` yields 7.67

Statistical functions

Function	Description
`mean(x, na.rm=FALSE)`	Arithmetic mean of object `x`
`sd(x)`	Standard deviation of object `x`
`sd(x)`	Variance of object `x`
`mad(x)`	Median absolute deviation of values in `x`
`median(x)`	Median value of object `x`
`quantile(x, probs)`	Quantiles where `x` is the numeric vector whose quantiles are desired and `probs` is a numeric vector with probabilities in $[0, 1]$
`range(x)`	Range
`sum(x)`	Sum
`min(x)`	Minimum
`max(x)`	Maximum

Note: most of these functions have many more parameters!

Exercises

1. Generate the following vectors in R using `seq`, `rep` and combinations thereof:

```
> c(4,5,6,4,5,6,4,5,6)
> c(1,1,1,1,2,2,2,2,3,3,3,3)
> c(0, 0.2, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0)
> c(1,1,2,2,3,3,8,9,8,9,8,9,8,9)
```

2. Consider the vector

```
> a = c(17.4, 19.4, 15.3, 19.3, 25.3, 16.4, 20.3, 18.6)
```

- Extract the 1st, 2nd and 4th elements and assign them to a new vector `b`
- Figure out how to calculate the length of a vector
- Extract the last 50% of the vector and assign it to a vector `c`

Logical operations

Given a vector we can apply logical operations to compare vector(s) or a vector with a scalar:

```
> x = c(1, 6, 2, 8, 10, 4, 3); y = c(9, 4, 2, 5, 2, 18, 4)
>
> x < y # strictly less
## [1]  TRUE FALSE FALSE FALSE FALSE  TRUE  TRUE

> x >= y # greater or equal
## [1] FALSE  TRUE  TRUE  TRUE  TRUE FALSE FALSE

> x == max(x) # equality
## [1] FALSE FALSE FALSE FALSE  TRUE FALSE FALSE

> !(x == max(x)) # negation
## [1]  TRUE  TRUE  TRUE  TRUE FALSE  TRUE  TRUE

> (x >= 3) & (x <= 5) # logical AND (component-wise)
## [1] FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE

> (y > 9) | (y < 3) # logical OR (component-wise)
## [1] FALSE FALSE  TRUE FALSE  TRUE  TRUE FALSE
```

Logical operations

Attention: for binary Boolean/logical operators there also exist "long" versions which only evaluate the first element!

```
> x = 1:4
> y = -1:2
> (x < 3) & (y < 2) # component-wise
## [1]  TRUE  TRUE FALSE FALSE

> (x < 3) && (y < 2) # only the first component is checked and no warning!

## Error in (x < 3) && (y < 2):  'length = 4' in coercion to 'logical(1)'

> all((x < 3) & (y < 2)) # are all component-wise comparisons true?
## [1] FALSE

> any(!((x < 3) & (y < 2))) # is at least one component-wise comparison false?
## [1] TRUE
```

Moreover, the longer version implements *short-circuit evaluation*, i.e., if we say `x && y` for two scalars `x` and `y`, and `x` evaluates to `TRUE`, `y` is no evaluated at all.[12]

[12] These semantics can by found in most languages.

Missing data

R is build around data and unfortunately data is often missing ☹
Missing data is represented with the keyword `NA` in vectors[13].

```
> age = c(24, 23, 21, 21, NA, 31, NA, 19)
> age
## [1] 24 23 21 21 NA 31 NA 19

> mean(age) # most functions return NA if there is at least one NA in the data
## [1] NA

> mean(age, na.rm = TRUE) # set na.rm = TRUE to skip NA values
## [1] 23.16667

> is.na(age) # # check if elements are NA (x == NA does not work!)
## [1] FALSE FALSE FALSE FALSE  TRUE FALSE  TRUE FALSE

> !is.na(age) # get only non-NA values
## [1]  TRUE  TRUE  TRUE  TRUE FALSE  TRUE FALSE  TRUE

> age[complete.cases(age)] # same
## [1] 24 23 21 21 31 19
```

[13] `NA` for Not Available.

Vector subsetting by indexing

Access a subset of the vector elements by integer indices:

```
> x = c(10, 6, 3, 6, 3, 5, 9, 5, 15, 6)
> x[5] # single elments
## [1] 3

> x[-5] # all but the 5th element
## [1] 10  6  3  6  5  9  5 15  6

> x[-c(5, 7)] # all but the 5th and 7th element
## [1] 10  6  3  6  5  5 15  6

> x[c(1, 5, 10)] # multiple elements
## [1] 10  3  6

> x[1:2] = 20 # re-assign the first two elements
> x
##  [1] 20 20  3  6  3  5  9  5 15  6
```

Exercises

1. Consider the vector $x = (49, 14, 25, 49, 14, 63, 65, 99, 56, 29)$. Create this vector in R
2. Create a logical vector where the i^{th} entry is `TRUE` if the i^{th} entry of `x` is equal to the minimum value of `x`
3. The function `rev` reverts the order of a vector. Use it to revert the first half of `x`
4. Normalize/rescale the vector: i.e., from each element subtract the minimum and divide by the difference of maximum and minimum value (functions `min` and `max` will be useful)

Vector subsetting by name

Vector elements can be named. The names can be used for subsetting:

```
> x = c("a" = 190, "b" = 20, "c" = 31)
> x[2] # by index of course still works
##  b
## 20

> x["a"] # single element
##   a
## 190

> ns = c("a", "b", rep("c", 3))
> x[ns]
##   a   b   c   c   c
## 190  20  31  31  31

> unname(ns) # drop names
## [1] "a" "b" "c" "c" "c"

> x["d"] # non-existing name -> NA
## <NA>
##   NA
```

Vector subsetting with logical values

Remember the logical operations? We can also use logical vectors to index vectors.

```
> x = c(10, 6, 3, 6, 3, 5, 9, 5, 15, 6)
>
> large = x >= 6
> large
##  [1]  TRUE  TRUE FALSE  TRUE FALSE FALSE  TRUE FALSE  TRUE  TRUE

> x[large]
## [1] 10  6  6  9 15  6

> x[x >= 6 & x <= 10]
## [1] 10  6  6  9  6

> x[rep(c(TRUE, FALSE), each = 5)]
## [1] 10  6  3  6  3
```

How to obtain indices of items of interest

Occasionally it is useful to not get the actual components but their indices. Here, function `which(x)` shines:

```
> x = c(10, 6, 3, 6, 3, 5, 9, 5, 15, 6)
>
> x >= mean(x)
##  [1]  TRUE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE  TRUE FALSE

> idx_large = which(x >= mean(x))
> idx_large
## [1] 1 7 9

> x[idx_large]
## [1] 10  9 15

> idx_min = which(x == min(x))
> idx_min
## [1] 3 5

> idx_min = which.min(x)
> idx_min
## [1] 3
```

Some very useful functions

Homework: toy around with these functions. Check the documentation (`?function`).

Function	Description
`sort(x)`	Sort elements
`order(x)`	Indices of elements in sorted order
`unique(x)`	Vector of unique elements (removes duplicates)
`duplicated(x)`	Which elements of `x` are duplicates?
`which.min(x)`	Index of smallest element
`which.max(x)`	Index of largest element
`which(x)`	Indices of elements in `x` which are `TRUE`

Exercises

1. Create a vector as follows:

```
> x = sample(1:5, size = 25, replace = TRUE)
```

2. Research what the function `sample` does.
3. Apply `duplicated(x)` and try to make sense out of it.
4. Create a vector which contains all unique elements of `x` in two different ways.
5. Generate a vector `y` which contains the indices of the duplicates of `x`.
6. Create a vector `y` which contains the indices of unique elements in the sorted version of `x`.

Attributes

- Every object in R can have *attributes* (bascially key-value pairs) attached. This is basically meta-data appended to the object.

```
> x = c(1:5, NA, NA, 6, 2, 6)
> attr(x, "len") = length(x)
> attr(x, "nna") = sum(is.na(x))
> x
##  [1]  1  2  3  4  5 NA NA  6  2  6
## attr(,"len")
## [1] 10
## attr(,"nna")
## [1] 2

> attr(x, "nna") # read attribute
## [1] 2

> attributes(x)  # read all attributes
## $len
## [1] 10
##
## $nna
## [1] 2

> attr(x, "len") = NULL # drop attribute
```

- Most attributes are dropped by operations (except for `dim` and/or `names`).

Matrices

A matrix in R is like a matrix in mathematics:

```
> x = matrix(1:9, ncol = 3)
> x
##      [,1] [,2] [,3]
## [1,]    1    4    7
## [2,]    2    5    8
## [3,]    3    6    9

> x[2, 3] # element in 2nd row and 3rd column
## [1] 8

> x[1, ] # first row (vector)
## [1] 1 4 7

> x[1, , drop = FALSE] # keep matrix structure even if we extract a single line
##      [,1] [,2] [,3]
## [1,]    1    4    7

> x[, 2] # first column (vector)
## [1] 4 5 6

> x[c(1, 3), 1:2] # sub-matrix
##      [,1] [,2]
## [1,]    1    4
## [2,]    3    6
```

Matrices

Actually a matrix is just a vector with a `dim` attribute:

```
> attributes(x)
## $dim
## [1] 3 3

> y = 1:9
> attr(y, "dim") = c(3, 3) # "manually" transform to matrix
> y
##      [,1] [,2] [,3]
## [1,]    1    4    7
## [2,]    2    5    8
## [3,]    3    6    9

> is.matrix(y)
## [1] TRUE
```

Matrices

Since matrices are vectors we can apply all functions that work for vectors:

```
> x - 10
##      [,1] [,2] [,3]
## [1,]   -9   -6   -3
## [2,]   -8   -5   -2
## [3,]   -7   -4   -1

> mean(x)
## [1] 5

> x[x >= 5] # subsetting drops dim attribute
## [1] 5 6 7 8 9

> x[x %% 2 == 0] = 100 # assignment does not drop
> x
##      [,1] [,2] [,3]
## [1,]    1  100    7
## [2,]  100    5  100
## [3,]    3  100    9
```

Functions on matrices

Function	Description
`A %*% B`	Matrix product $A \cdot B$
`A * B`	Element-wise product of matrices $(A_{ij} \cdot B_{ij}, 1 \leq i, j, \leq n)$
`t(A)`	Matrix transposition
`eigen(A)`	Eigenvalues and Eigenvectors of A
`solve(A)`	(regular) inverse A^{-1} of A
`solve(A, b)`	Solution x of equation $Ax = b$
`rowMeans(A)`	Row-wise mean values
`colMeans(A)`	Column-wise mean values
`rowSums(A)`	Row-wise sums
`colSums(A)`	Column-wise sums
`nrow(A), ncol(A)`	Number of rows/columns of A
`dim(A)`	Dimension (i.e., rows and columns) of A
`diag(A)`	Vector of diagonal elements of A

Lists

Lists are similar to vectors, but each element can have a different data type/class.

```
> l = list(name = "Jakob", lectures = c("DA1", "OR"), age = 37)
>
> l[[1]] # access element by index
## [1] "Jakob"

> l$name # access by name
## [1] "Jakob"

> l[["name"]] # alternative (good if name is stored in variable)
## [1] "Jakob"

> l[["name"]] = "Jakob B." # assignment works too
> l[["name"]]
## [1] "Jakob B."

> l[["lectures"]][1]
## [1] "DA1"
```

Often the return values of complex functions are named lists
$\rightsquigarrow$ multiple return values.

Lists (cont.)

We can add elements to existing lists:

```
> l = list(name = "Jakob", lectures = c("DA1", "OR"), age = 37)
> l$surname = "Bossek"
> l
## $name
## [1] "Jakob"
##
## $lectures
## [1] "DA1" "OR"
##
## $age
## [1] 37
##
## $surname
## [1] "Bossek"
```

Concatenation works as well:

```
> l2 = c(l, list(degree = "PHD", studied = "computer science"))
> is.list(l2)
## [1] TRUE


> l2$degree
## [1] "PHD"
```

Dataframes

Data frames are lists where every components has the same length:

```
> x = data.frame(
+   id = 1:4,
+   name = c("Max", "Sophie", "Jack", "Ted"),
+   grade = c(5.0, 5.0, 4.0, 5.0))
> x
##   id   name grade
## 1  1    Max     5
## 2  2 Sophie     5
## 3  3   Jack     4
## 4  4    Ted     5

> x$name
## [1] "Max"    "Sophie" "Jack"   "Ted"

> x[1:2, c("name", "id")] # equal to x[1:2, c(2, 1)]
##     name id
## 1    Max  1
## 2 Sophie  2

> x[1:2, ]$id # get id (alernatives: x[1:2, "id"] or x[1:2, 1])
## [1] 1 2
```

Dataframes: subsetting

```
> # ...
> x$sex = c("M", "F", "M", "M") # add another variable
> head(x, n = 3) # show first n rows only
##   id   name grade sex
## 1  1    Max     5   M
## 2  2 Sophie     5   F
## 3  3   Jack     4   M

> tail(x, n = 1) # show last n rows only
##   id name grade sex
## 4  4  Ted     5   M

> # Subsetting works the way we expect it to work
> x[x$sex == "M", ] # get all males
##   id name grade sex
## 1  1  Max     5   M
## 3  3 Jack     4   M
## 4  4  Ted     5   M

> x[(x$sex == "M") & (x$grade != 5), "name"] # get all males that failed the DA1 exam :(
## [1] "Jack"
```

Dataframes: subsetting

```
> # ...
> x1 = x[(x$sex == "M") & (x$grade != 5), "name"] # Remember this?
>
> subset(x, sex == "M" & grade != 5, select = "name") # same result, nicer interface
##   name
## 3 Jack

> # Some helpful functions (also work for matrices)
> ncol(x) # number of columns
## [1] 4

> nrow(x) # number of rows
## [1] 4

> dim(x)  # dimension (nr. of rows, nr. of columns)
## [1] 4 4
```

Dataframes: within

The `within` enables to evaluate an expression in an environment constructed from data.[14]

```
> data(airquality) # check docs with ?airquality
> head(airquality, n = 1)
##   Ozone Solar.R Wind Temp Month Day
## 1    41     190  7.4   67     5   1
```

We can modify the data as follows (hard to read)

```
> airquality$Month = factor(month.abb[airquality$Month])
> airquality$TempCel = round((airquality$Temp - 32) * 5/9, 1) # Fahrenheit to Celsius
> airquality$Day = NULL
```

Alternative with `within`

```
> data(airquality)
> airquality = within(airquality, {
+   Month = factor(month.abb[Month])
+   TempCel = round((Temp - 32) * 5/9, 1) # Fahrenheit to Celsius
+   rm(Day)
+ })
```

[14] This is not limited to dataframes.

Dataframes: import

Importing data from flat files in CSV-format:[15]

```
> # writing to a csv file using . as decimal point and ; as the "comma"
> data(mtcars)
> write.table(mtcars, file = "mtcars.csv", row.names = TRUE, sep = ";", dec = ".")
```

The content of `mtcars.csv`:

```
"mpg";"cyl";"disp";"hp";"drat";"wt";"qsec";"vs";"am";"gear";"carb"
"Mazda RX4";21;6;160;110;3.9;2.62;16.46;0;1;4;4
"Mazda RX4 Wag";21;6;160;110;3.9;2.875;17.02;0;1;4;4
...
```

```
> # now let us import the data
> mtcars = read.table("mtcars.csv", header = TRUE, sep = ";")
> head(mtcars , 2)
##                 mpg cyl disp  hp drat    wt  qsec vs am gear carb
## Mazda RX4        21   6  160 110  3.9 2.620 16.46  0  1    4    4
## Mazda RX4 Wag    21   6  160 110  3.9 2.875 17.02  0  1    4    4
```

15 CSV = Comma Separated Values. Very common flat-file format for rectangular data.

Dataframes: combining

```
> names = c("Anton", "Jack", "Tobias", "Sophie")
> ages = c(21, 31, 26, 35)
>
> cbind(names, ages) # cbind = column bind
##      names    ages
## [1,] "Anton"  "21"
## [2,] "Jack"   "31"
## [3,] "Tobias" "26"
## [4,] "Sophie" "35"

> as.data.frame(cbind(names, ages))
##    names ages
## 1  Anton   21
## 2   Jack   31
## 3 Tobias   26
## 4 Sophie   35

> stud1 = list(name = "Anton", age = 21)
> stud2 = list(name = "Sophie", age = 35)
>
> rbind(stud1, stud2) # rbind = row bind
##       name     age
## stud1 "Anton"  21
## stud2 "Sophie" 35
```

Data frames: aggregating

In data analysis we often split a data frame by some variables and calculate summary statistics, e.g., the mean, variance etc.

```
> data(mtcars)
>
> aggregate(
+   mtcars[, c("mpg", "cyl", "disp", "gear")], # data set
+   by = list(mtcars$cyl), # list of grouping variables
+   FUN = mean # aggregation function
+ )
##   Group.1      mpg cyl     disp     gear
## 1       4 26.66364   4 105.1364 4.090909
## 2       6 19.74286   6 183.3143 3.857143
## 3       8 15.10000   8 353.1000 3.285714
```

Data frames: aggregating

Splitting by multiple variables:

```
> aggregate(
+   mtcars[, c("mpg", "cyl", "disp", "gear")], # data set
+   by = list(mtcars$cyl, mtcars$gear), # grouping variable
+   FUN = function(x) {
+     c(mean = mean(x), sd = sd(x))
+   },
+   drop = TRUE # drop unused combinations of grouping values
+ )
##   Group.1 Group.2   mpg.mean     mpg.sd cyl.mean cyl.sd  disp.mean    disp.sd
## 1       4       3 21.5000000         NA        4     NA 120.100000         NA
## 2       6       3 19.7500000  2.3334524        6      0 241.500000  23.334524
## 3       8       3 15.0500000  2.7743959        8      0 357.616667  71.823494
## 4       4       4 26.9250000  4.8073604        4      0 102.625000  30.742699
## 5       6       4 19.7500000  1.5524175        6      0 163.800000   4.387862
## 6       4       5 28.2000000  3.1112698        4      0 107.700000  17.819091
## 7       6       5 19.7000000         NA        6     NA 145.000000         NA
## 8       8       5 15.4000000  0.5656854        8      0 326.000000  35.355339
##   gear.mean gear.sd
## 1         3      NA
## 2         3       0
## 3         3       0
## 4         4       0
## 5         4       0
## 6         5       0
## 7         5      NA
```

Data frames: outlook

R ships with nice methods for subsetting (`subset`), aggregating (`aggregate`) etc., but data analysis become real fun with additional packages:

```
> suppressPackageStartupMessages(library(tidyverse))
> mtcars %>%
+   select(mpg, cyl, disp, gear) %>%
+   group_by(cyl, gear) %>%
+   summarize_all(list(mean = mean, sd = sd)) %>%
+   ungroup()
## # A tibble: 8 x 6
##     cyl  gear mpg_mean disp_mean mpg_sd disp_sd
##   <dbl> <dbl>    <dbl>     <dbl>  <dbl>   <dbl>
## 1     4     3     21.5      120. NA       NA
## 2     4     4     26.9      103.  4.81    30.7
## 3     4     5     28.2      108.  3.11    17.8
## 4     6     3     19.8      242.  2.33    23.3
## 5     6     4     19.8      164.  1.55     4.39
## 6     6     5     19.7      145  NA       NA
## 7     8     3     15.0      358.  2.77    71.8
## 8     8     5     15.4      326   0.566   35.4
```

We will learn about the `tidyverse` suite next time!

Exercises

1. Load the `mtcars` data set (`data(mtcars)`)
2. Subset all observations where the horsepower is among the hightest 50% of the values
3. Subset all observations where the weight is in the interval $[2, 3]$ and the number of cylinders is at least 6
4. Create a data frame that contains the first and last 5 observations of `mtcars`
5. Research how to create a random subset of observations (Hint: check `?sample`)
6. Add a new variable `wtcat` which takes two character values: "heavy" if the weight exceeds 4 and "light" otherwise.

Factors aka categorical variables

Oftentimes variables have a limited number of catgories, e.g., sex, field of study or, say, experience with R.

```
> sex_char = c("F", "M", "F", "F", "M")
> sex_char
## [1] "F" "M" "F" "F" "M"

> sex_fac = factor(sex_char) # convert to factor
> sex_fac
## [1] F M F F M
## Levels: F M

> sex_fac[2] = "F"
> sex_fac[2] = "other" # fails!

## Warning in `[<-.factor`(`*tmp*`, 2, value = "other"):  invalid factor level, NA generated

> sex_char[2] = "other" # no problem at all
> sex_char
## [1] "F"     "other" "F"     "F"     "M"
```

Factors: renaming categories

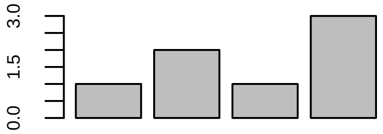
Sometimes it makes sense to rename factor levels:

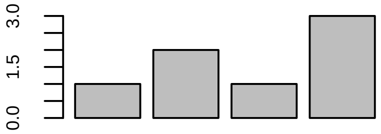
```
> # no explict levels
> sex_fac = factor(c("F", "M", "F", "F", "M"))
> sex_fac
## [1] F M F F M
## Levels: F M

> # level = category
> levels(sex_fac)
## [1] "F" "M"

> # we can also use it to rename
> levels(sex_fac) = c("Female", "Male", "Other")
> sex_fac
## [1] Female Male   Female Female Male
## Levels: Female Male Other

> sex_fac[2] = "Other"
> sex_fac
## [1] Female Other  Female Female Male
## Levels: Female Male Other
```

Factors: example

Imagine we imported data where missing values were encoded differently (not the R way with `NA`):

```
> # A dash "-" indicates a missing value
> party = factor(c("SPD", "CDU", "CDU", "SPD", "-", "SPD", "Grüne"))
> # The dash is a factor level now
> party
## [1] SPD   CDU   CDU   SPD   -     SPD   Grüne
## Levels: - CDU Grüne SPD

> opar = par(mfrow = c(1, 2), cex.axis = 0.6) # some graphical parameter
> plot(party)
> levels(party)[1] = "Missing"
> plot(party)
```





Exercises

1. Create the following vector where both a dash and and empty string represent missing values:

```
> x = c("a", "b", "c", "-", "", "c", "a", "b", "c", "-", "/")
```

2. Use logical expressions to replace all occurrences of "-" and "" with `NA` in `x`
3. Convert `x` to a factor
4. Plot a barplot. What do you observe? Is this behaviour always desireable?

Ordered factors

Sometimes categories can be ordered, e.g., for *knowledge in R* we may have *good* $>$ *medium* $>$ *none*.

- We can tell R that a factor is indeed ordered.
- Used by plots, e.g., to arrange factor levels on axis respecting their order.[16]

```
> r_fac = factor(c("good", "medium", "medium", "none", "good"),
+   levels = c("good", "medium", "none"), ordered = TRUE)
> r_fac
## [1] good   medium medium none   good
## Levels: good < medium < none

> # comparison makes sense for ordered factors ...
> r_fac[1] < r_fac[2]
## [1] TRUE

> # ... but not for unordered factors
> sex_fac[1] > sex_fac[2]
## [1] NA
```

16 Default is alphabetic order.

Exercises

1. Load the mtcars dataset via `data(mtcars)`
2. Read the documentation of the data set (`?mtcars`)
3. Plot a histogram of the gross horsepower variable
4. Now we want to add a new variable `power` to the data. Use the function `cut` to convert gross horsepower to the categorical variable `power` with factor levels $(0, 150]$, $(150, 250]$ and $(250, 350]$ (see argument `breaks` of `cut`)
5. Plot a barplot of `power`
6. Rename the factor levels to *low*, *medium* and *high* and make the factor ordered

Inspecting objects

Very helpful function `str` for structure gives a nice overview of the data types/classes of an object (applicable to all kinds of R objects):

```
> str(x)
## 'data.frame':  4 obs. of  4 variables:
##  $ id   : int  1 2 3 4
##  $ name : chr  "Max" "Sophie" "Jack" "Ted"
##  $ grade: num  5 5 4 5
##  $ sex  : chr  "M" "F" "M" "M"

> str(1:4)
##  int [1:4] 1 2 3 4

> str(factor(c("good", "bad", "bad", "bad"), levels = c("good", "bad"), ordered = TRUE))
##  Ord.factor w/ 2 levels "good"<"bad": 1 2 2 2

> str(mean)
## function (x, ...)
```

Output

Executing R scripts

If we have an R script, say `myscript.R`, with R code we can execute it in an interactive session by typing

```
> source("myscript.R")
```

We can also execute scripts from the command line. This is powerful if you want to, e.g.,

- Automate processes or
- Call R through other tools.

```
Rscript myscript.R
Rscript myscript.R > output.txt # redirect output to file (on unix only)
R -e 'install.packages("ggplot2", dependencies = TRUE)'
```

Output

So far we just wrote the name of an object and it got printed to the console, e.g.:

```
> x = 10
> x
## [1] 10
```

This works perfectly find in *interactive mode*, but not if we run a script via `Rscript myscript.R` or `source("myscript.R")`. In these cases cases like the 2nd line in the above listing will have no effect.
The following works as expected though:

```
> x = 10
> print(x)
## [1] 10
```

Output: string formatting

Function `sprintf(fmt, ...)` expects a so-called *format-specifier*[17] and an arbitrary number of arguments that are parsed/used by the former.

```
> sprintf("Just a string.") # works perfectly fine
## [1] "Just a string."

> x = 10
> y = 19.45353325
> s = "DA1 is awesome!"
> sprintf("x has the value %i", x) # %i for integer
## [1] "x has the value 10"

> sprintf("x = %i, y = %f", x, y) # %f for fixed point decimal notation [-]mmm.ddd
## [1] "x = 10, y = 19.453533"

> sprintf("y = %.2f", y) # two decimal number after .
## [1] "y = 19.45"

> sprintf("Simple fact: %s", s) # %s for character
## [1] "Simple fact: DA1 is awesome!"
```

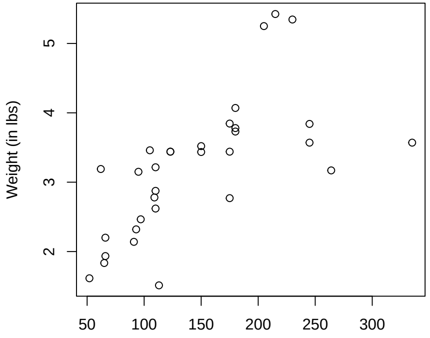
17 Essentially a character with specific instructions embedded. The latter are substituted by the formatted arguments passed after the format string in order of appearance.

Basic R graphics

Basic R graphics: plot

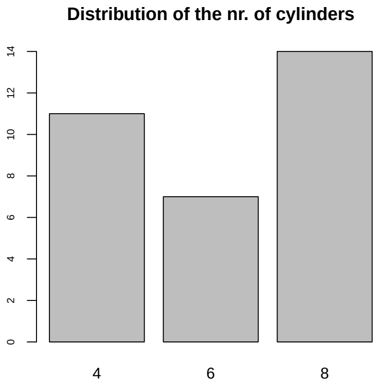
R offers heaps of build-in plot functionality:

```
> data(mtcars)
> plot(mtcars$hp, mtcars$wt, xlab = "Gross horsepower", ylab = "Weight (in lbs)")
```



Basic R graphics: barplot

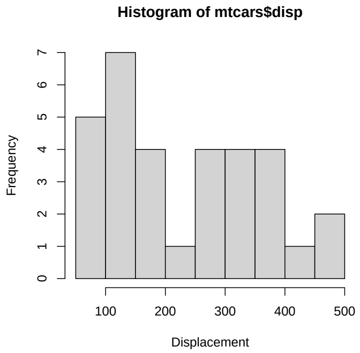
```
> tab = table(mtcars$cyl)
> barplot(tab, cex.axis = 0.7, main = "Distribution of the nr. of cylinders")
```



Basic R graphics: hist

For univariate numeric variables the histogram is a good starting point:

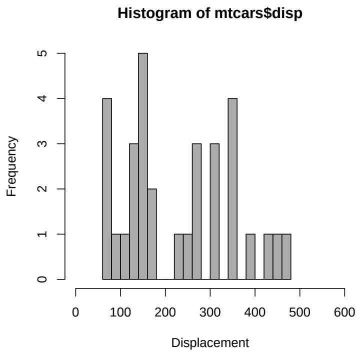
```
> hist(mtcars$disp, xlab = "Displacement")
```



Basic R graphics: hist

For univariate numeric variables the histogram is a good starting point:

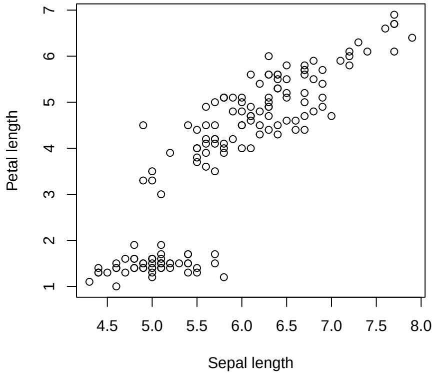
```
> hist(mtcars$disp, breaks = 20, freq = TRUE, col = "grey67",
+   xlim = c(0, 600), xlab = "Displacement")
```



Basic R graphics: scatter-plot

For bivariate numeric observations:

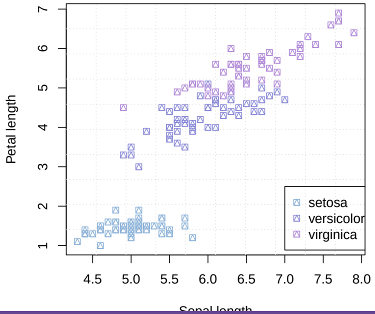
```
> data(iris)
> plot(iris$Sepal.Length, iris$Petal.Length,
+   xlab = "Sepal length", ylab = "Petal length")
```



Basic R graphics: scatter-plot

For bivariate numeric observations:

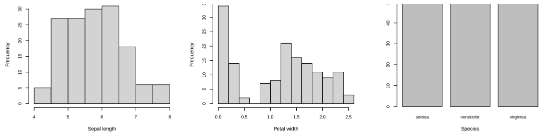
```
> cols = c("#8DB3D9", "#8D8DD9", "#B38DD9") # colors in RGB format
> plot(iris$Sepal.Length, iris$Petal.Length,
+   xlab = "Sepal length", ylab = "Petal length",
+   col = cols[iris$Species], pch = 14) # color by species (factor)
> grid(nx = 10, ny = 10, col = "lightgray", lty = "dotted", equilogs = TRUE)
> legend(7, 2.5, legend = c("setosa","versicolor","virginica"),
+   col = cols, pch = 14)
```



Basic R graphics: matrix of plots

Function par serves to either query or set graphical parameters which are applied to all subsequently produced plots.

```
> # save old settings in opar
> opar = par(
+   mfrow = c(1, 3), # (1 x 3) matrix arrangement of plots
+   cex.axis = 0.9, # axis labels font size slightly reduced
+   oma = rep(0, 4) # size of outer margins
+ )
> hist(iris$Sepal.Length, xlab = "Sepal length")
> hist(iris$Petal.Width, xlab = "Petal width")
> barplot(table(iris$Species), xlab = "Species")
```



```
> par(opar) # reset settings
```

Basic R graphics: saving plots

Plots can be exported in the most prominent formats: PDF[18], PNG[19], TIFF[20] etc.

```
> png("myfile.png", units = "cm", width = 8, height = 10, res = 72) # initialize output
> barplot(iris$Species)
> dev.off() # close device. I.e., write file.
>
> jpeg("myfile.jpeg", quality = 85) # quality adjusts JPG compression
> boxplot(iris$Sepal.Width ~ iris$Species)
> dev.off()
>
> # works also for collections
> pdf("myfile.pdf", width = 700, height = 400)
> opar = par(
+   mfrow = c(1, 2), # (1 x 2) matrix arrangement of plots
+ )
> hist(iris$Sepal.Length, xlab = "Sepal length")
> hist(iris$Petal.Width, xlab = "Petal width")
> par(opar) # reset settings
> dev.off()
```

18 PDF = Portable Data Format; vector-based format.

19 PNG = Portable Network Graphics; pixel-based format.

20 TIFF = Tag Image File Format.

Exercises

1. Install and load the `ggplot2` package with

```
> install.packages("ggplot2", dep = TRUE)
> library(ggplot2)
```

2. Load the diamonds dataset via `data(diamonds)`

3. Take a random sample of 500 diamonds:[21]

```
> x = diamonds[sample(1:nrow(diamonds), size = 500), ]
```

4. Toy around with the learned visualizations. Some ideas:

   - Generate histograms of `price` and or `carat` side by side.
   - Draw scatter-plots of `x` vs. `y` or `price` vs. `carat`. In the latter plot color the points by `cut` and add a legend; figure out how to omit the border of the legend.

[21] The data set contains 53 940 observations. It may take some time to render plots for that many observations. Try it!

Basic R graphics: conclusion

- Simple, yet powerful.
- We just scratched the surface here!
  $\rightsquigarrow$ Essentially we can achieve anything with base R graphics.
- However, some actions are not directly supported:
  - Facetting (multiple panels).
  - Error bars.
  - Legends need to be crafted manually.
  - etc.
  - Pro: generation of base R graphics is fast.
- Later we will learn about `ggplot` (Wickham 2009), a sophisticated visualization library/package.[22]

[22] It is not the holy grail, but way better and visually appealing.

Control flow

Conditional statements

Conditions are elemantary building blocks of (procedural) programming: do something if certain logical conditions hold and do something else if they do not hold.

```
> x = c(1, 2, 3, 3, 2, 1, 5, 3)
> (x %% 2) # modulo operator -> rest of integer division with 2
## [1] 1 0 1 1 0 1 1 1

> (x %% 2) == 0 # ith entry is TRUE if ith number is even
## [1] FALSE  TRUE FALSE FALSE  TRUE FALSE FALSE FALSE

> all((x %% 2) == 0) # TRUE if all elements of logical vector are TRUE
## [1] FALSE

> if (all((x %% 2) == 0)) {
+   "all numbers even"
+ } else {
+   "at least one number is odd"
+ }
## [1] "at least one number is odd"
```

Conditional statements

If, else if, . . . , else if, else:

```
> x = c(43.3, 15.3, NA, NA, 34.3, Inf, 19.3, 15.4)
>
> xna = is.na(x)
> xinf = is.infinite(x)
>
> if (any(xna) & any(xinf)) {
+   # print "prints" its argument to the console even if the script is not executed
+   # interactively, but is loaded from an R-script
+   print("There are NAs and infinite values")
+ } else if ((sum(xna) > 1) & (all(x < 40))) {
+   print("There is at least one NA and all values re at most 40")
+ } else {
+   print("Nothing interesting to report")
+ }
## [1] "There are NAs and infinite values"
```

Conditional statements: invalid inputs

The condition needs to be or evaluate to a single Boolean value!

```
> if ("char") 1

## Error in if ("char") 1:  argument is not interpretable as logical

> if (c()) 1

## Error in if (c()) 1:  argument is of length zero

> if (mean(c(1, NA, 4, 3) > 2)) 1

## Error in if (mean(c(1, NA, 4, 3) > 2)) 1:  argument is not interpretable as logical

> if (c(TRUE, FALSE, TRUE)) 1

## Error in if (c(TRUE, FALSE, TRUE)) 1:  the condition has length > 1
```

Since R 3.5.0. we can turn the latter to throw an error via

```
> Sys.setenv("_R_CHECK_LENGTH_1_CONDITION_" = "true") # defaults to "false"
> if (c(TRUE, FALSE)) 1

## Error in if (c(TRUE, FALSE)) 1:  the condition has length > 1
```

Conditional statements: vectorized if

The last example on the previous slide is actually not constructed at all: imagine we want to modify each element of a vector if some condition on this very element holds:

```
> x = c(6, 2, 6, 3, 2, 6, 2, 3, 8, 9, 1)
>
> # categorize into odd/even numbers (the laborious way)
> y = x
> for (i in seq_along(y)) {
+   if (x[i] %% 2 == 0) {
+     y[i] = "even"
+   } else {
+     y[i] = "odd"
+   }
+ }
> y
##  [1] "even" "even" "even" "odd"  "even" "even" "even" "odd"  "even" "odd"
## [11] "odd"

> # and the R way
> y = ifelse(x %% 2 == 0, "even", "odd")
```

For loop

Repetitive tasks are tedious. Programming languages like them a lot! ☺

```
> for (i in 1:3) {
+   print(i^2)
+ }
## [1] 1
## [1] 4
## [1] 9

> l = list(a = c(1, 3, 4), b = c(4, 5, NA), c = c(7,3,6))
> names(l) # returns the names
## [1] "a" "b" "c"

> means = list()
> for (name in names(l)) {
+   means[[name]] = mean(l[[name]], na.rm = TRUE)
+ }
> unlist(means) # "flatten" the list
##        a        b        c
## 2.666667 4.500000 5.333333
```

For loop

It is usually better to use `seq_along` to avoid hard to find errors if the sequence we iterate over is empty:

```
> x = c(2, 5, 2, 3, 3, 5)
> x = x[x > 10] # empty vector
>
> # Bad
> for (i in 1:length(x)) { # length(x) is 0
+   paste0(i, ": ", x[i]) # concatenate elements to character
+ }
>
> # Good: use seq_along(...)
> for (i in seq_along(x)) { # length(x) is 0
+   paste0(i, ": ", x[i]) # concatenate elements to character
+ }
>
> # for illustration
> 1:c() # not what we would like to have in a for loop

## Error in 1:c():  argument of length 0

> seq_along(c()) # this look better
## integer(0)
```

For loop: next and break

We can skip iterations or break the entire loop earlier if some condition is satisfied:

```
> for (i in c(1, 2, NA, 4:10)) {
+   if (is.na(i))
+     next

+   print(i)

+   if (i >= 8)
+     break
+ }
## [1] 1
## [1] 2
## [1] 4
## [1] 5
## [1] 6
## [1] 7
## [1] 8
```

While loop

Repeat something as long as a logical condition is (not) met:

```
> # In this situation a for loop would be the better choice
> x = matrix(runif(100), nrow = 5)
> i = 1
>
> while (i <= nrow(x)) {
+   mean(x[i, ])
+   i = i + 1 # need to increase manually. Otherwise: infinite loop
+ }
>
> x = runif(1)
> while(x > 0.1) { # sample random number in [0,1] until a number < 0.1 is sampled
+   x = runif(1)
+ }
```

Loops are slow in R, aren't they?

Oftentimes you will read: avoid loops in R. They are slow as hell!

Golden rules to not slow down your R code with loops

- Use R's *vectorization*. I.e., if there exists a build-in vectorized function, use it!
- Avoid using `c` or `cbind/rbind` in loops to grow objects
  $\rightsquigarrow$ R needs to re-allocate memory in every iteration.[23]
- Instead, try to pre-allocate memory in advance.

Still a loop is unavoidable?

- Allocate memory in advance, e.g. `x = numeric(1000)` if you know that you will populate 1 000 positions of a numeric vector.
- If it is really a bottleneck $\rightsquigarrow$ implement function in C or C++.

[23] If allocation takes linear time, a linear number of re-allocations takes qaudratic time.

R loops vs. build-in functions: study I

Study: Given a length *n* vector *x* let us build a vector *y* which contains the *cumulative sum*,[24] i.e.,

$$y_i = \sum_{j=1}^{i} x_i, i = 1, \ldots, n.$$

```
> x = rnorm(100000) # data generation: N(0, 1), i.e., normally distributed randoms
> n = length(x)
>
> system.time(cumsum(x)) # build-in C-based function (ultra fast)
##    user  system elapsed
##       0       0       0

> system.time({ # BAD: growing vector in each iteration (ultra slow)
+   cs = c(x[1])
+   for (i in 2:n) {
+     cs = c(cs, cs[i - 1] + x[i])
+   }
+ })
##    user  system elapsed
##   7.101   1.179   8.297
```

[24] Note that calculations are not independent here, so vectorization is not an option.

R loops vs. build-in functions: study I

Here are better loop solutions using pre-allocated memory:

```
> system.time({ # GOOD: pre-allocate memory in advance (good performance)
+   cs = numeric(n)
+   cs[1] = x[1]
+   for (i in 2:n) {
+     cs[i] = cs[i - 1] + x[i]
+   }
+ })
##    user  system elapsed
##   0.005   0.000   0.005

> system.time({
+   cs = numeric(n)
+   cs[1] = x[1]
+   i = 2
+   while (i <= n) {
+     cs[i] = cs[i - 1] + x[i]
+     i = i + 1
+   }
+ })
##    user  system elapsed
##   0.007   0.000   0.008
```

R loops vs. build-in functions: study II

Now let us check the performance of vectorizable operations. For a given vector x we want to calculate a simple linear transformation $a \cdot x + b$ for some constants $a, b \in \mathbb{R}$:

```
> x = runif(1000000) # uniform andom numbers in [0, 1]
> n = length(x)
> a = 10
> b = 100
>
> system.time({y = a * x + b}) # build-in C-based function (ultra fast)
##    user  system elapsed
##   0.001   0.001   0.001

> system.time({y = sapply(x, function(e) a * e + b)}) # vectorized (slower than loop!)
##    user  system elapsed
##   0.490   0.021   0.512

> system.time({ # Not bad at all
+   y = x
+   for (i in 1:n) {
+     y[i] = a * y[i] + b
+   }
+ })
##    user  system elapsed
##   0.038   0.001   0.039
```

Functions

Reusable code

Functions are the most basic building blocks if it comes to reusable code. For those who are not familiar with the concept: imagine we want to calculate $\sum_{i=1}^{n} x_i \cdot y_i$ over and over again for different x and y.

```
> x = 1:10; y = 11:20
> s = 0
> # this is actually a bad example for a loop, but helps to state the problem here
> for (i in 1:10) {
+   s = s + (x[i] * y[i])
+ }
>
> x = c(10, 24, 53); y = c(29, 10, 4)
> s = 0
> for (i in 1:10) {
+   s = s + (x[i] * y[i])
+ }
>
> # and so on ...
```

Pretty tedious, error-prone and redundant isn't it?

Functions

$$\text{Input arguments} \longrightarrow \boxed{\text{Function}} \longrightarrow \text{Return value(s)}$$

- Functions encapusulate code that solves some interesting sub-task and is general enough to be reused.
- Allow for automation of common tasks.
- It is much like a math. function. E.g.

$$f : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R} \text{ with } f(x,y) \mapsto \sum_{i=1}^{n} x_i \cdot y_i .$$

- The good thing is: once we have a useful function, we no longer care about its exact implementation, but only about the function *interface/signature*. I.e., what does the function expect as input and what does it return as output?
- R functions are objects!

  $\rightsquigarrow$ Functions can be assigned to variables, stored in vectors, passed down to other

Functions: definition

Let's write a function for our problem: $\sum_{i=1}^{n} x_i \cdot y_i$.

```
> sumprod = function(x, y) {
+   r = sum(x * y)
+   return(r)
+ }
>
> sumprod(c(29, 10, 4), c(10, 24, 53))
## [1] 742

> sumprod(x = 1:10, y = 11:20)
## [1] 935

> sumprod(x = c(10, 24, 53), c(29, 10, 4))
## [1] 742

> sumprod(y = c(29, 10, 4), x = c(10, 24, 53))
## [1] 742
```

Functions: scoping (simplified)

Scoping = process of finding the value associated with a name.
In functions we can (and often do[25]) redefine objects / variables names; the inner-most definition is relevant.

```
> x = 1;y = 10
> f = function() {
+   x = 100 # this variable "masks" the definition outside the function
+   c(x, y)
+ }
> f()
## [1] 100  10

> x
## [1] 1

> f = function() {
+   x = x + 1
+   x
+ }
> f()
## [1] 2

> x
## [1] 1
```

Lazy evaluation

Lazy evaluation means that an expression is evaluated only if the expression (calculation, value of object) is actually used.

```
> lazyfun = function(x, y) {
+   if (x < 10)
+     x + 10
+   else
+     x + y[1]
+ }
>
> lazyfun(10) # works since y is never used and thus not evaluated in lazyfun

## Error in lazyfun(10):  argument "y" is missing, with no default
```

Powerful tool! Helps to avoid costly calculations if the result won't be used.

```
> lazyfun(4)
## [1] 14

> lazyfun(10) # y not passed, but function tries to use it

## Error in lazyfun(10):  argument "y" is missing, with no default
```

Lazy evaluation

The concept of lazy evaluation is a powerful tool! It helps to avoid costly calculations if the result(s) won't be used.

```
> lazyfun = function(x, y) {
+   if (x < 10)
+     x + 10
+   else
+     x + y # y is only touched if x is smaller 10
+ }
>
> lazyfun(4) # works since y is never used and thus not evaluated in lazyfun
## [1] 14

> lazyfun(20) # errors, since y not passed, but function tries to use it

## Error in lazyfun(20):  argument "y" is missing, with no default

> system.time({lazyfun(4, var(runif(100000000)))}) # costly evaluation skipped
##    user  system elapsed
##       0       0       0

> system.time({lazyfun(20, var(runif(100000000)))}) # costly evaluation not skipped
##    user  system elapsed
##   0.799   0.092   0.894
```

Lazy evaluation

Lazy evaluation is not limited to functions! It is a language concept.

```
> if (TRUE) {
+   print(100)
+ } else {
+   print(z) # z not defined, but this branch is never executed
+ }
## [1] 100

> x = 1:1000000
> if (x[1] == 1 || (sqrt(sum(x^2)) > 500)) { # costly second condition skipped
+   print("Second condition not checked.")
+ }
## [1] "Second condition not checked."
```

Functions: invocation

The usual way to call a function is by using the familiar notation. E.g.:

```
> x = runif(10) # some random numbers
> mean(x, na.rm = TRUE, trim = 0.1)
## [1] 0.6346447
```

Occasionally it happens that our arguments are already stored in a list:

```
> args = list(x, na.rm = TRUE, trim = 0.1)
> do.call(mean, args) # call mean fun with parameter given in args
## [1] 0.6346447
```

Helpful to glue together dataframes:

```
> x = data.frame(x = 1, y = "a")
> y = data.frame(x = 2, y = "b")
> do.call(rbind, list(x, y))
##   x y
## 1 1 a
## 2 2 b
```

Functions: composition

Often we apply a sequence of functions to data:

1. **Nesting**: useful for short sequences.

```
> x = runif(10)
> y = runif(10)
> sqrt(sum((x - y)^2))
## [1] 1.360139
```

2. **Intermediate objects**: can become tedious.

```
> tmp = (x - y)^2
> tmp = sum(tmp)
> sqrt(tmp)
## [1] 1.360139
```

3. **Piping**:[26] using the magnificent `magrittr` package.[27]

```
> library(magrittr)
> (x-y)^2 %>%
+   sum() %>%
+   sqrt()
## [1] 1.360139
```

[26] Similar to pipe operator in unix command line.

[27] We will make extensive use of piping later.

Exercises

1. Write a function `means(x)` which expects a numeric matrix `x` and returns a vector where the i^{th} entry corresponds to the arithmetic mean of the i^{th} column of x.
2. Modify `means` such that it expects another argument `of`. If `of="column"` the function shall behave as in (1). If `of="row"` the row-wise means shall be calculated instead.
3. The Fibonacci-numbers[28] is a famous recursive infinite sequence of numbers following the building rule

$$F_1 = 0, F_2 = 1 \text{ and } F_n = F_{n-1} + F_{n-2} \text{ for } n \geq 3.$$

   Write a function `fibonacci(n)` which returns a vector of the first n Fibonacci numbers.

28 https://en.wikipedia.org/wiki/Fibonacci_number

The *apply-function family

Often we have an n-dim. object $x = (x_1, x_2, \ldots, x_n) \in \mathcal{D}^n$ and a function $f : \mathcal{D} \to \mathcal{E}$ that we want to apply to each element of x:

$$f(x) = f((x_1, x_2, \ldots, x_n)) \mapsto (f(x_1), f(x_2), \ldots, f(x_n)) .$$

E.g.

- Apply a function to each element of a vector.
- Apply a function to each column (or row) of a matrix.
- Apply a function to each matrix in a list.
- etc.

The *apply-function family

`lapply(data, function)` is the most generic: given a vector/list `data` it applies function `function` to every component and returns a list or vector:

```
> x = list(a = 1:10, b = 10:20)
>
> # in order to calculate the the maximum of each component we could do
> x_max = numeric(2)
> for (i in 1:2) {
+   x_max[i] = max(x[[i]])
+ }
> x_max
## [1] 10 20

> lapply(x, max) # readable alternative
## $a
## [1] 10
##
## $b
## [1] 20

> # here we pass an "anonymous" function to calculate the range
> lapply(x, function(e) max(e) - min(e))
## $a
## [1] 9
##
## $b
```

The *apply-function family

`apply(data, MARGIN, function)` works on matrices. The second parameter `MARGIN` indicates if the function should be applied row- or colwise:

```
> x = matrix(1:9, ncol = 3)
> x
##      [,1] [,2] [,3]
## [1,]    1    4    7
## [2,]    2    5    8
## [3,]    3    6    9

> apply(x, 1, sum) # rowwise sum
## [1] 12 15 18

> apply(x, 2, var) # colwise variance
## [1] 1 1 1
```

Code style

Actually, the R interpreter does not care whether your code looks like this ...

```
> x=c(24, 23, 21, 21, NA, 31, NA, 19)
> f2=function(x, na.rm=FALSE){
+ if(na.rm){x = x[complete.cases(x)]}
+ return(sum(x)/length(x))}
> f(x)
```

... or ...

```
> age = c(24, 23, 21, 21, NA, 31, NA, 19)
> average = function(x, na.rm = FALSE) {
+   if(na.rm) { # drop NAs if argument is TRUE
+     x = x[complete.cases(x)]
+   }
+   return(sum(x) / length(x))
+ }
> average(age)
```

But you should care! ☺

⤳ Use e.g., Google's R Style Guide[29] for your own good!

[29] `https://google.github.io/styleguide/Rguide.html`

Some Dos and Don'ts

- Use a style guide.
- Use speaking names for objects[30]:

```
> x = c(24, 25, 23, 31, 54, 31) # bad
> ages = c(24, 25, 23, 31, 54, 31) # good
> f = function(x) { ... } # bad
> rotate_vector = function(x, theta) { ... } # good
```

- Avoid overwriting existing objects, e.g., by defining a variable `mean` or a function `c`.
- Document your code:

```
> k = 3 # initialize variable k to 3 (BAAAD!)
> k = 3 # number of clusters for k-means algorithm (good)
>
> # ToDo: add documentation (BAAAD! :))
> rotate_vector = function(x, theta) { ... }
>
> # Rotates a numeric vector counterclockwise by a given angle.
> # @param x [numeric] Numeric input vector of length at least 2.
> # @param theta [numeric] Rotation angle in degrees.
> # @return [numeric] Rotated vector x. (GOOD!)
> rotate_vector = function(x, theta) { ... }
```

30 I admit we did not do it here very often. We used rather short names to save precious space.

Packages

. . . are collections[31] of functions and data.
Packages can be installed from the Comprehensive R Archive Network (CRAN) via

```
> install.packages("devtools", dep = TRUE)
```

Development versions, e.g., from Git(Hub) repositories can be installed via

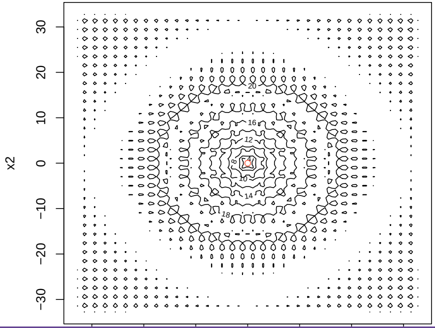
```
> devtools::install_github("jakobbossek/smoof", dep = TRUE)
```

31 Basically software libraries.

Packages (cont.)

Once a package is installed it can be loaded and all exported functions and datasets can be used:

```
> library(smoof)
> # the prefix 'smoof::' is not necessary, but useful to keep track where the function
> # is implemented
> fn = smoof::makeAckleyFunction(dimensions = 2)
> plot(fn, show.optimum = TRUE, render.levels = FALSE)
```



Advanced: Parallelization

Parallelization in R: The Sad News

One of R's drawbacks: R will use one core regardless of how many cores the CPU exposes.

How to deal with it?

- Simply use another programming language, e. g., Java, C, ...
- Interface R from `C` or `Python`.
- Compile the 64Bit Version using special build options.
- Use nice packages developed by the R community (see CRAN HPC task View).

Types of parallelism

Implicit parallelism parallelism hidden from user, system abstracts it away.

Explicit parallelism user need to handle paralellism explicitely by using special directives.

Implicit Parallelism

Implicit parallelism?

Parallelism is hidden from the user, no user-requests/directives needed. Thus parallelism is automatically exploited.

- Paralellized computation of mathematical functions in Blas library.
- Installation on windows systems is simple: replace Rblas.dll with compiled one.
- Installation on unix systems: configure R with `-with-blas` flag and compile afterwards; fastest implementation is OpenBlas (hand optimized assembler code).

Explicit Parallelism

Explicit parallelism?

Parallelism is activated by the user calling specific directives.

Drawbacks and pitfalls

- User needs to care for distribution of objects, loading of packages in the slave jobs.
- Logging is difficult.
- Identifying errors is even harder than in sequential mode.
- Random Number Generators (RNGs) must be initialized occasionally (this is of utmost importance in statisticl computing).

Explicit Parallelism: parallel

First simple package:

- Package shipped with R since version 2.14.0.
- Slight extension of `snow` and `multicore`.
- Includes parallel Random Numbers Generators (RNG).
- Main entry point is parallel version of `lapply`.
- Single CPU usage (`multicore`) or several machines (`snow`).
- On windows only in socket mode ☹.

Explicit Parallelism: parallel (cont.)

```
> library(parallel)
> n.cores = 2 # nuumber of cores
>
> x = 1:100
> x = split(x, rep (1:n.cores, each = length(x) / n.cores))
> str(x)
## List of 2
##  $ 1: int [1:50] 1 2 3 4 5 6 7 8 9 10 ...
##  $ 2: int [1:50] 51 52 53 54 55 56 57 58 59 60 ...

> y = unlist(mclapply(x, sum)) # parallel sum
> print(y)
##    1    2
## 1275 3775

> sum(y)
## [1] 5050
```

Explicit Parallelism: parallel (cont.)

Example: *k*-means clustering on `mtcars` data.[32]

```
> f = function(i) {
+   suppressPackageStartupMessages(library(mlr)) # load mlr package
+   lrn = makeLearner("cluster.kmeans", centers = 2) # define k-means algorithm
+   data(mtcars, package = "datasets")
+   cluster.task = makeClusterTask(data = mtcars)
+   mod = train(lrn, task = cluster.task) # calculate clustering
+ }
> n.cores = detectCores()
> system.time({lapply(1:n.cores, f)}) # not parallel

## Error in requirePackages(package, why = stri_paste("learner", id, sep = " "), :  For learner cluster.kmeans please install the
following packages:  clue
## Timing stopped at:  0.319 0.032 0.955

> system.time({mclapply(1:n.cores, f)}) # parallel

## Warning in mclapply(1:n.cores, f):  all scheduled cores encountered errors in user code
##    user  system elapsed
##   0.001   0.023   0.023
```

[32] The difference in runtime is almost neglegible in this example since `mtcars` is a very small data set. However, if on larger data one run of *k*-means takes a minute you will experience a massive, close to optimal speed-up if you perform at most as many runs as there are CPU cores.

Explicit parallelism: parallelMap

- Major drawback of methods seen so far: backend change requires change of code in general ☹
- R package `parallelMap`[33] is another wrapper
- Single function to learn: `parallelMap`
- No need to change code if backend changes
- Configurable via options
- Support for most important parallelization modes, i. e., multicore machines, socket mode, MPI and HPC

33 https://github.com/berndbischl/parallelMap

Explicit parallelism: parallelMap (cont.)

```
> library(parallelMap)
> # start in socket mode
> parallelStartSocket(cpus = 2L)

## Starting parallelization in mode=socket with cpus=2.

> f = function(i) {
+   res = summary(runif(1e6))
+ }
> parallelMap(f, 1:2) # like lapply(1:2, f)

## Mapping in parallel: mode = socket; level = NA; cpus = 2; elements = 2.
## [[1]]
##      Min.   1st Qu.    Median      Mean   3rd Qu.      Max.
## 0.0000001 0.2502242 0.4998253 0.4997979 0.7497259 0.9999988
##
## [[2]]
##      Min.   1st Qu.    Median      Mean   3rd Qu.      Max.
## 0.0000022 0.2496570 0.4994759 0.4995947 0.7492900 0.9999994

> parallelStop()

## Stopped parallelization. All cleaned up.
```

Topics not covered

These slides just scratch the surface:

- The many peculiarities of R.[34]
- Object-oriented programming (S3, S4, R6).
- Metaprogramming.
- Advanced functional programming.
- Literate programming for writing reports and presentations where text, code and evaluation are intermingled.
- Interfacing C natively or C++ via `Rcpp`.
- Package development.

Work through Advanced R by H. Wickham (Wickham 2014) if you are interested in more details.

[34] IMHO it is not useful to teach this. These are things that come with experience.

Literature Recommendations

Literature Recommendations

1. Gareth James, Daniela Witten, Trevor John Hastie & Robert Tibshirani (2021). An Introduction to Statistical Learning. 2nd Edition. Springer. (James et al. 2013)★
   $\rightsquigarrow$ Brief R intro; R labs at the end of each chapter.
2. Hadley Wickham & Garrett Grolemund (2017). R for Data Science: Import, Tidy, Transform, Visualize, and Model Data. O'Reilly. (Wickham and Grolemund 2017)
3. Hadley Wickham (2016). ggplot2: Elegant Graphics for Data Analysis. Springer. (Wickham 2009)
4. Hadley Wickham (2014). Advanced R. Chapman & Hall/CRC The R Series. Taylor & Francis. (Wickham 2014)
5. Hadley Wickham (2015). R Packages. 1st Edition. O'Reilly Media. (Wickham 2015)

What comes next?

Overview of the upcoming week:

Date	**Content**	**Speaker**
21 Oct.	Brief introduction to R ✓	Jakob
22 Oct.	Tutorial on math foundations	Jakob
28 Oct.	Preprocessing of data (with R)	Jakob
29 Oct.	Tutorial on R	Raphael
⋮	⋮	⋮

Wrap-Up

Wrap-Up

Todays content

R introduction (basic concepts, loops, functions etc.)

Your task(s)

- Download and install both R[35] and R Studio[36].
- Work through the R-intro in James et al. 2013.
- Work through the presentation slides and do the many exercises.
- Work on exercise sheet 02.

[35] URL: `https://www.r-project.org`
[36] URL: `https://www.rstudio.com`

References I

Wickham, Hadley (2009). *ggplot2: elegant graphics for data analysis.* Springer New York. ISBN: 978-0-387-98140-6.

— (2014). *Advanced R.* Chapman & Hall/CRC The R Series. Taylor & Francis. ISBN: 9781466586963. URL: `https://books.google.de/books?id=PFHFNAEACAAJ`.

James, Gareth et al. (2013). *An Introduction to Statistical Learning: with Applications in R.* Springer.

Wickham, Hadley and Garrett Grolemund (Jan. 2017). *R for Data Science: Import, Tidy, Transform, Visualize, and Model Data.* 1st ed. O'Reilly Media. ISBN: 1491910399.

Wickham, Hadley (2015). *R Packages.* 1st. O'Reilly Media, Inc. ISBN: 1491910593.