# 02 - Introduction to R & RStudio

## Visual Analytics Exercise

### Summer Term

# What is R?

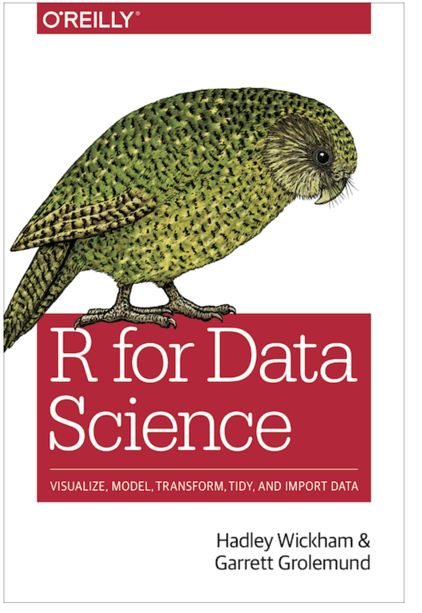
- statistical programming language which was develloped especially for data analysis tasks
- free, open source and available on every major OS
- passed the mark of 10,000 packages in January 2017
    - numerous packages for statistics and machine learning
    - elaborate packages for creating graphics and charts
- IDE made for interactive (visual) data analysis and statistical programming

Figure source: Wikipedia. *R (programming language)*. Accessed 19.02.2018. URL: https://en.wikipedia.org/wiki/R_(programming_language))

# Help & Materials

Helpful links for getting started:

- The Comprehensive R Archive Network (CRAN)
- Major R IDE: RStudio
- Aggregators of R news and tutorials: R-Bloggers, R Weekly
- Book R for Data Science by Hadley Wickham (2017)
- Book Hands-On Programming with R by Garrett Grolemund (2014)
- Free online class Introduction to R on DataCamp
- Learn R interactively with swirl
- Online reference explaining the most important functions for data import, basic statistics and chart creation: Quick-R
- "Cheat sheets" for R: R Reference Card, RStudio Cheat Sheets



# R IDE: RStudio

Most important panes:

- Console
- Code
- Environment
- History
- Plots
- Files
- Packages
- Help

RStudio Overview - 1:30 from RStudio, Inc. on Vimeo.

# Basic commands 1/2

Run simple commands:

```
3 + 4
```

```
## [1] 7
```

Assign values to variables:

```
x ← 3
y ← 4
x + y
```

```
## [1] 7
```

# Basic commands 2/2

Create vectors:

```
c(1,2,3) # combine elements into a vector
```

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

```
1:3 # an integer sequence
```

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

```
seq(1,3, by = 0.5) # a sequence of values from 1 to 3 with an 0.5 increment
```

```
## [1] 1.0 1.5 2.0 2.5 3.0
```

```
rep(1:3, times = 2) # repeat a vector
```

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

```
rep(1:3, each = 2) # repeat each element of a vector
```

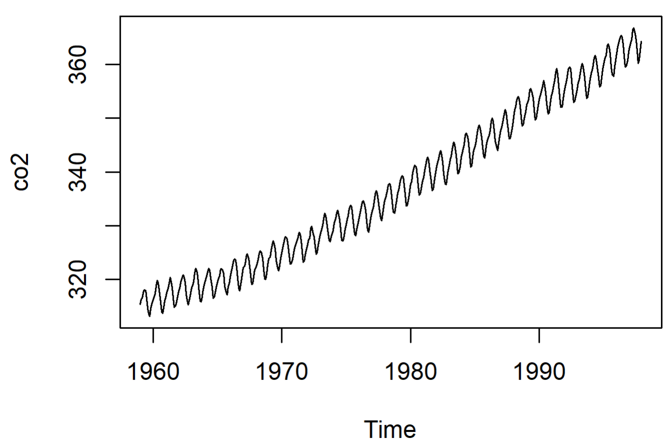
```
## [1] 1 1 2 2 3 3
```

# The R Ecosystem

- The CRAN version of R (*Base R*) contains the scaffolding of essential functions and several datasets for learning data analysis
- List all available datasets with `data()` (*Note: `data()` is a function*)
- Display a function's code with `data` (No parenthesis)
- Create simple diagrams with `graphics::plot()`

Example: atmospheric CO2 concentration at *Mauna Loa* (active vulcane on Hawaii) over time

```
plot(co2)
```



# R Packages

- *base R* contains functions that are needed by the majority of the users
- more specific functions can be included *on demand* by loading **packages**
- a package is a collection of functions, data, and documentation that extends the capabilities of base R

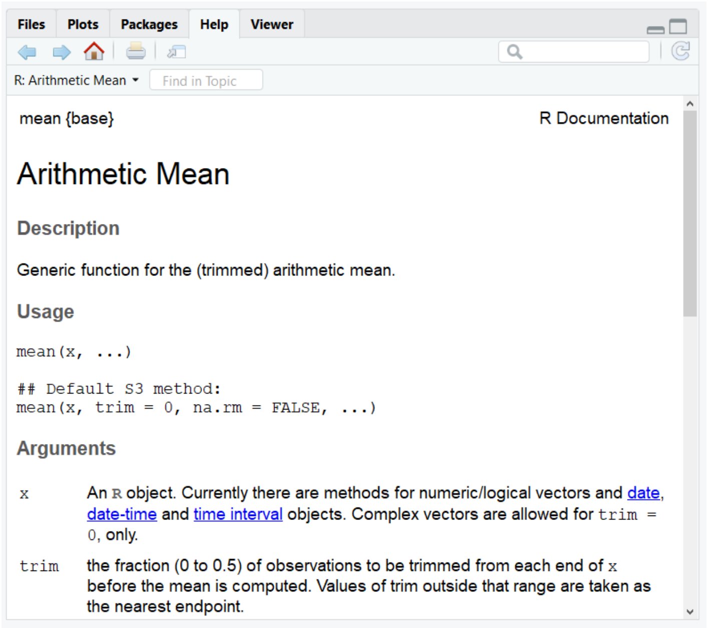
Download, install and load a package from CRAN:

```
# A package has to be installed just once, but ...
install.packages("dplyr")
#  ... it has to be loaded in each session
# where functions of this package are needed.
library(dplyr)
```

# Help

Get help of a particular function:

```
?mean
help(mean) # does the same
```



# Comments

```
# This is a comment. This line doesn't get executed.
```

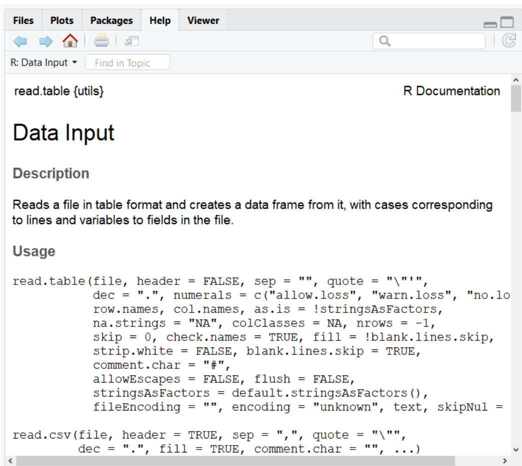
# Data Analysis - Introductory example

> Imagine that we had been running a pedestrian survey in the USA. We had let pedestrians write their height and sex on a sheet of paper. Afterwards, the handwritten statements were digitalized and saved in a csv file. Our task is to summarize and describe the heights of the survey participants to a person not involved in the study.

# Read files 1/2

First, the dataset has to be loaded into the R environment, e.g. using the `read.table()` function.

```
?read.table
```




> **Description** Reads a file in table format and creates a data frame from it, with cases corresponding to lines and variables to fields in the file.

We have to specify the `file` argument.

> **file** the name of the file which the data are to be read from.

# Read files 2/2

Download file from lab website to hard drive and load local file in R using `read.csv`.

```
# getwd() # Return working directory.
# ⟶ R searches for files here by default.
# setwd() # Change working directory.
# ⟶ Alternatively in RStudio via 'Session'>'Set Working Directory'
getwd()
```

```
## [1] "C:/Users/benedikt/Docume ... 18/02_Introduction_to_R"
```

```
# We store the dataset in the subfolder 'Data'
dat <- read.csv("Data/heights.csv")
```

# Glimpse at the dataset

Quickly examine the object `dat`:

```
str(dat) # str stands for structure
```

```
## 'data.frame':    148 obs. of  3 variables:
## $ Timestamp : chr "1/25/2016 8:15:15" "1/25/2016 8:15:21" "1/25/2016
##    8:15:25" "1/25/2016 8:15:29" ...
## $ What.is.your.gender.  : chr "Female" "Female" "Male" "Female"  ...
## $ What.is.your.height..in.inches..: chr "63" "62" "69" "68"  ...
```

The object is a `data.frame`. Generally, data frames are used to store tables.

To see `dat` in table form and individual values, we can use `View()`:

```
View(dat)
```

| | Timestamp | What.is.your.gender. | What.is.your.height..in.inches.. |
|---|---|---|---|
| 1 | 1/25/2016 8:15:15 | Female | 63 |
| 2 | 1/25/2016 8:15:21 | Female | 62 |
| 3 | 1/25/2016 8:15:25 | Male | 69 |
| 4 | 1/25/2016 8:15:29 | Female | 68 |
| 5 | 1/25/2016 8:15:37 | Male | 71.65 |
| 6 | 1/25/2016 8:15:37 | Male | 75 |
| 7 | 1/25/2016 8:15:39 | Male | 68.8976 |
| 8 | 1/25/2016 8:15:40 | Male | 74 |

Showing 1 to 8 of 148 entries

Previous 1 2 3 4 5 … 19 Next

# Extract data frame columns and vector elements

Extract columns with `$` or `[[]]`:

```
dat$Timestamp # or
dat[["Timestamp"]] # or
dat[[1]] # index
```

The output is a vector. To access values of a vector, the operator `[]` is used:

```
dat$Timestamp[2]
```

```
## [1] "1/25/2016 8:15:21"
```

# Fundamental vector functions

```
sort(dat[[3]]) # sort heights
```

```
##  [1] "167"           "168"          "172"          "180"
##  [5] "180"           "180"          "5' 11\""      "5' 7\""
##  [9] "5'10"          "5'10"         "5'10''"       "5'10\""
## [13] "5'11"          "5'11\""       "5'2"          "5'4"
## [17] "5'5"           "5'6"          "5'7"          "5'7"
## [21] "5'7\""         "5'8"          "5'8\""        "5 ft 9 inches"
## [25] "5ft 9 inches"  "6'1\""        "60"           "60"
## [29] "60"            "60"           "60"           "60"
## [33] "61"            "61"           "61"           "61"
## [37] "62"            "62"           "62"           "62"
```

```
unique(dat[[2]]) # unique genders
```

```
## [1] "Female"                   "Male"
## [3] "I prefer not to disclose"
```

```
table(dat[[2]]) # contingency table on gender
```

```
##
##                   Female I prefer not to disclose                     Male
##                       68                        1                       79
```

# Vectors in R 1/4

In `R`, there are 2 types of vectors: **atomic vectors** and **lists**. Lists are recursive vectors, i.e., they can contain other lists.

Atomic Vectors are sequences of elements that are homogeneous w.r.t. their type. The most important **atomic data types** in R are `logical`, `integer`, `double` and `character`.

**C**ombine elements to vectors with `c()`:

```
x <- c(1,2,3,4,5)
x
```

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

... or by using `:` or `seq()`:

```
x <- 1:5
x <- seq(1,5)
```

# Vectors in R 2/4

Vectors have two key properties: type and length.

```
typeof(x)
```

```
## [1] "double"
```

```
length(x)
```

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

## Name/rename vector elements:

```
names(x) <- letters[1:5]
x
```

```
## a b c d e
## 1 2 3 4 5
```

```
typeof(x)
```

```
## [1] "double"
```

# Vectors in R 3/4

Access vector elements:

```
x[2] # the second element
```

```
## b
## 2
```

```
x["b"] # element with the name "b"
```

```
## b
## 2
```

```
x[-2] # all but the second element
```

```
## a c d e
## 1 3 4 5
```

```
x[-(3:5)] # all elements except three to five
```

```
## a b
## 1 2
```

# Vectors in R 4/4

```
x[c(1,5)] # elements one and five
```

```
## a e
## 1 5
```

```
x[x == 2] # elements which are equal to 2
```

```
## b
## 2
```

```
x == 2
```

```
##     a     b     c     d     e
## FALSE  TRUE FALSE FALSE FALSE
```

```
x < 5
```

```
##     a     b     c     d     e
##  TRUE  TRUE  TRUE  TRUE FALSE
```

```
x[x %in% c(1,4,5)] # elements in the set 1,4,5
```

```
## a d e
## 1 4 5
```

# Vector type hierarchy in R

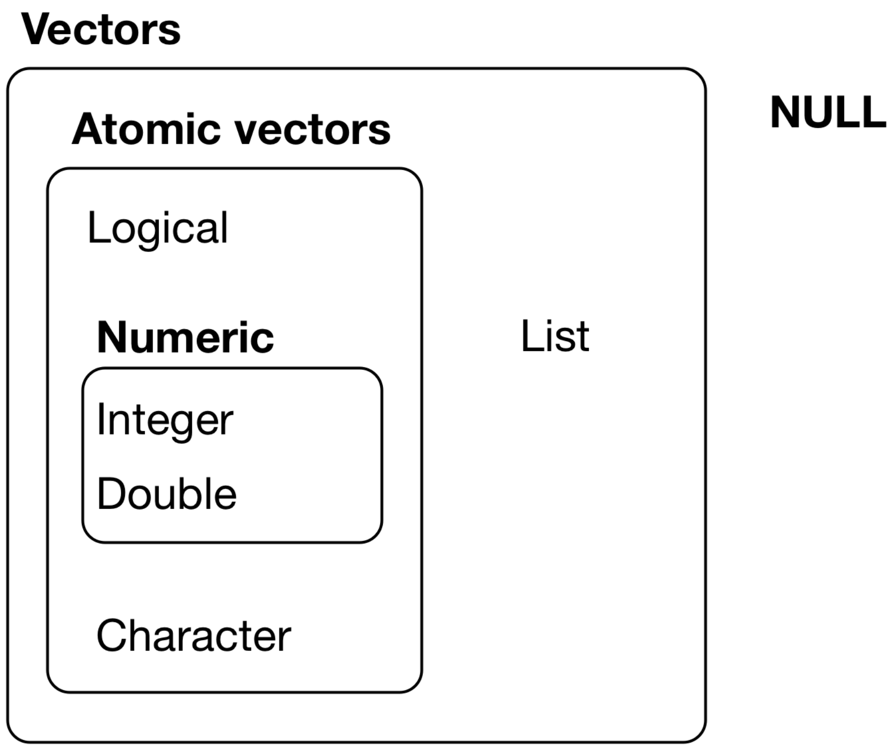



Figure source: Hadley Wickham and Garrett Grolemund. *R for Data Science*. O'Reilly, 2017.

# Vector coercion in R

- a vector must be homogeneous w.r.t. the data type of its elements
- R tries to **coerce** (~*erzwingen*) the values into the most flexible type
- thus, creating a vector with elements of different types doesn't yield an error

**Order of types** from **least flexible** to **most flexible**:

`logical` ⟶ `integer` ⟶ `double` ⟶ `character`.

```
height <- c(60, 59, 55, "5'5", 70) # doesn't yield an error
str(height)
```

```
## chr [1:5] "60" "59" "55" "5'5" ...
```

Coercion can be useful when working with functions:

```
mean(c(FALSE, FALSE, TRUE)) # Proportion of 'TRUE' entries
```

```
## [1] 0.3333333
```

# Vectorization

```r
height <- c(60, 59, 55, "5'5", 70)
```

What is the data type of the vector?

```r
height[3]
```

```
## [1] "55"
```

Convert character to numeric value:

```r
as.numeric(height[3])
```

```
## [1] 55
```

One of R's big advantages is that most operations can be *vectorized*.

```r
as.numeric(height)
```

```
## Warning: NAs durch Umwandlung erzeugt
```

```
## [1] 60 59 55 NA 70
```

This yields a warning. R doesn't know how it should convert "5'5" into a number.

# Factors 1/3

- special class of vector which takes only pre-defined values
- predominantly used for categorical variables
- the pre-defined values of a factor are called **levels**

```
x <- factor(c("f", "m", "m", "f"))
x
```

```
## [1] f m m f
## Levels: f m
```

```
class(x)
```

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

```
levels(x)
```

```
## [1] "f" "m"
```

# Factors 2/3

Values which are not in the set of levels **cannot be used**.

```
x[2] <- "a"
```

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

```
x
```

```
## [1] f    <NA> m    f
## Levels: f m
```

# Factors 3/3

Factors are particularly useful if all possible values are known beforehand, even when they don't occur at first.

```
sex_char <- c("m", "m", "m")
sex_factor <- factor(sex_char, levels = c("m", "f"))

table(sex_char)
```

```
## sex_char
## m
## 3
```

```
table(sex_factor)
```

```
## sex_factor
## m f
## 3 0
```

# Data manipulation with `dplyr`

> R comprises powerful and flexible functions and operators for data manipulation. However, sometimes the syntax is hard to follow. Using the `dplyr` package single data manipulation steps can be formulated with a syntax that is similar to the English language.
>
> `dplyr` is part of a package collection called *tidyverse*. "All packages share an underlying design philosophy, grammar and data structures."[1]

```
library(dplyr)
library(readr) # Functions for data import
# We use the faster 'read_csv()' instead of 'read.csv()'
# 'read_csv' doesn't convert character vectors
# automatically into factors (different to 'read.csv()')
dat <- read_csv("Data/heights.csv")
```

```
##
## -- Column specification --------------------------------------------------------
## cols(
##   Timestamp = col_character(),
##   `What is your gender?` = col_character(),
##   `What is your height (in inches)?` = col_character()
## )
```

[1] See https://www.tidyverse.org/.

# Tibbles

The `dat` object is now of type `tbl_df`, a modification of `data.frame` with functions for a prettier display.

```
class(dat)
```

```
## [1] "spec_tbl_df" "tbl_df"      "tbl"         "data.frame"
```

```
dat
```

```
## # A tibble: 148 x 3
##    Timestamp         `What is your gender~ `What is your height (in inches~
##    <chr>             <chr>                 <chr>
##  1 1/25/2016 8:15:15 Female                63
##  2 1/25/2016 8:15:21 Female                62
##  3 1/25/2016 8:15:25 Male                  69
##  4 1/25/2016 8:15:29 Female                68
##  5 1/25/2016 8:15:37 Male                  71.65
##  6 1/25/2016 8:15:37 Male                  75
##  7 1/25/2016 8:15:39 Male                  68.8976
##  8 1/25/2016 8:15:40 Male                  74
##  9 1/25/2016 8:15:41 Female                65
## 10 1/25/2016 8:15:44 Female                5'4
## # … with 138 more rows
```

# Select columns

... using the `dplyr`-verb `select()`:

```
select(dat, contains("height"))
```

```
## # A tibble: 148 x 1
##    `What is your height (in inches)?`
##    <chr>
##  1 63
##  2 62
##  3 69
##  4 68
##  5 71.65
##  6 75
##  7 68.8976
##  8 74
##  9 65
## 10 5'4
## # ... with 138 more rows
```

```
# base R equivalent:
# dat[, 3]
# dat$`What is your height (in inches)?`
```

# Rename columns

The column names are rather verboose at the moment.

```
names(dat)
```

```
## [1] "Timestamp"                        "What is your gender?"
## [3] "What is your height (in inches)?"
```

Rename them:

```
names(dat) <- c("time", "sex", "height")
```

# Missing Values

Missing Values are displayed as `NA`. Missing values can be located with `is.na()`.

```
is.na(as.numeric(height))
```

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

Other special values include:

- `NaN` (not a number): e.g. `sqrt(-2)` $\longrightarrow$ `is.nan()`
- `Inf`: e.g. `1/0` $\longrightarrow$ `is.infinite()`
- `NULL`: absence of a whole vector $\longrightarrow$ `is.null()`

# Adding columns

... using the `dplyr` verb `mutate()`:

```
dat <- mutate(dat,
              numeric_height = as.numeric(height),
              original = height)

# Base R equivalent:
# dat$numeric_height <- as.numeric(height)
# dat$original <- height
```

# Filter observations

... using the `dplyr` verb `filter()`:

```
filter(dat, is.na(numeric_height))
```

```
## # A tibble: 21 x 5
##    time              sex                     height  numeric_height original
##    <chr>             <chr>                   <chr>            <dbl> <chr>
##  1 1/25/2016 8:15:44 Female                  "5'4"               NA "5'4"
##  2 1/25/2016 8:15:45 Female                  "5'8\""             NA "5'8\""
##  3 1/25/2016 8:15:45 Female                  "5'5"               NA "5'5"
##  4 1/25/2016 8:29:00 Male                    "5'7"               NA "5'7"
##  5 1/25/2016 14:39:~ Female                  "5'6"               NA "5'6"
##  6 1/25/2016 22:02:~ Male                    "5'11\~             NA "5'11\""
##  7 1/26/2016 8:36:33 I prefer not to discl~ "5'10\~             NA "5'10\""
##  8 1/26/2016 9:49:15 Male                    "5'7\""             NA "5'7\""
##  9 1/26/2016 9:49:19 Male                    "5'7"               NA "5'7"
## 10 1/26/2016 9:51:19 Female                  "5'8"               NA "5'8"
## # ... with 11 more rows
```

```
# Base R equivalent:
# dat[is.na(numeric_height), ]
```

# The pipe operator `%>%` 1/2

The **pipe operator `%>%`** helps to write **code that is easy to read and understand**. Its main purpose is to **express a sequence of multiple operations in a single step**. The value of the first argument of a function is the output of the last function before the pipe operator.

```
dat %>%
  filter(is.na(numeric_height)) %>%
  select(height) %>%
  slice(1:21)
```

```
## # A tibble: 21 x 1
##    height
##    <chr>
##  1 "5'4"
##  2 "5'8\""
##  3 "5'5"
##  4 "5'7"
##  5 "5'6"
##  6 "5'11\""
##  7 "5'10\""
##  8 "5'7\""
##  9 "5'7"
## 10 "5'8"
## # … with 11 more rows
```

More thorough explanation:

- Hadley Wickham and Garrett Grolemund. *R for Data Science - Chapter 14: Pipes with magrittr*. O'Reilly, 2017.
- Documentation of the `magrittr` package

# The pipe operator `%>%` 2/2

Compare...

```
# dplyr approach
dat %>%
  filter(is.na(numeric_height)) %>%
  select(height) %>%
  slice(1:21)
```

... with ...

```
# base R approach 1: each step is saved as new object
dat1 <- dat[is.na(dat$numeric_height), ]
dat1[1:21, "height"]
```

... or ...

```
# base R approach 2: one-liner
dat[which(is.na(dat$numeric_height))[1:21], "height"]
```

# String replacement with `str_replace()` 1/2

To convert the `height` variable to numeric, all non-numeric characters must be replaced.

The function `str_replace` from the `stringr` package searches for specific characters of strings and replaces them by others. More precisely, it detects a regular expression and replaces it by another regular expression.

```
x ← dat$height[109:116]
x
```

```
## [1] "5'10"          "70"            "67.7"          "62"
## [5] "5ft 9 inches"  "5 ft 9 inches" "5'2"           "74"
```

```
library(stringr)
# Each "ft" is replaced with "'".
x ← str_replace(x, "ft", "'")
x
```

```
## [1] "5'10"         "70"           "67.7"         "62"
## [5] "5' 9 inches"  "5 ' 9 inches" "5'2"          "74"
```

# String replacement with `str_replace()` 2/2

```
# Each "inches" is removed.
x <- str_replace(x, "inches", "")
x
```

```
## [1] "5'10"   "70"     "67.7"   "62"     "5' 9 "  "5 ' 9 " "5'2"    "74"
```

- replace all occurences of `"ft"` with `"'"`
- remove all occurences of quotation marks `"`, `"inches"`, spaces and inverted commas `'`

```
dat <- dat %>%
  mutate(height = str_replace(height, "ft", "'")) %>%
  # We remove all occurences of '"', 'inches', ' ' (space) and '''.
  mutate(height = str_replace_all(height, "\"|inches|\ |''", ""))
```

| | time | sex | height | numeric_height | original |
|---|---|---|---|---|---|
| 1 | 1/25/2016 8:15:15 | Female | 63 | 63 | 63 |
| 2 | 1/25/2016 8:15:21 | Female | 62 | 62 | 62 |
| 3 | 1/25/2016 8:15:25 | Male | 69 | 69 | 69 |
| 4 | 1/25/2016 8:15:29 | Female | 68 | 68 | 68 |
| 5 | 1/25/2016 8:15:37 | Male | 71.65 | 71.65 | 71.65 |
| 6 | 1/25/2016 8:15:37 | Male | 75 | 75 | 75 |
| 7 | 1/25/2016 8:15:39 | Male | 68.8976 | 68.8976 | 68.8976 |
| 8 | 1/25/2016 8:15:40 | Male | 74 | 74 | 74 |
| 9 | 1/25/2016 8:15:41 | Female | 65 | 65 | 65 |
| 10 | 1/25/2016 8:15:44 | Female | 5'4 | | 5'4 |

Showing 1 to 10 of 148 entries

Previous 1 2 3 4 5 … 15 Next

# Functions 1/5

Example function for calculating the arithmetic mean of a vector:

```r
avg <- function(x){
  return(sum(x) / length(x))
}
avg(1:5)
```

```
## [1] 3
```

Example function which calculates the variance of a vector:

```r
myVariance <- function(x) {
  return(sum((x - avg(x))^2)/length(x))
}
myVariance(1:5)
```

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

# Functions 2/5

Define a function which converts `5'4` to `5*12+4`:

```
library(purrr)
fixheight <- function(x){
  y <- str_split(x, "'")
  ret <- map_dbl(y, function(z){
    ifelse(length(z) > 1,
           as.numeric(z[1])*12 + as.numeric(z[2]) ,
           as.numeric(z[1]))
  })
  return(ret)
}
```

# Functions 2/5

Define a function which converts 5'4 to `5*12+4`:

```
library(purrr)
fixheight <- function(x){
  y <- str_split(x, "'")
  ret <- map_dbl(y, function(z){
    ifelse(length(z) > 1,
           as.numeric(z[1])*12 + as.numeric(z[2]) ,
           as.numeric(z[1]))
  })
  return(ret)
}
```

- The function `str_split()` partitions a character vector at the position of the split string and returns a **list** of substrings.

# Functions 2/5

Define a function which converts 5'4 to `5*12+4`:

```
library(purrr)
fixheight <- function(x){
  y <- str_split(x, "'")
  ret <- map_dbl(y, function(z){
    ifelse(length(z) > 1,
           as.numeric(z[1])*12 + as.numeric(z[2]) ,
           as.numeric(z[1]))
  })
  return(ret)
}
```

- The function `str_split()` partitions a character vector at the position of the split string and returns a **list** of substrings.
- The function `map_dbl()` from the `purrr` package[1] applies a function to each element of a vector and returns a vector of the same length and of type `double`.

[1] Hadley Wickham and Garrett Grolemund. *R for Data Science - Chapter 21: Iteration*. O'Reilly, 2017.

# Functions 3/5

Define a function which converts 5'4 to `5*12+4`:

```
library(purrr)
fixheight <- function(x){
  y <- str_split(x, "'")
  ret <- map_dbl(y, function(z){
    ifelse(length(z) > 1,
           as.numeric(z[1])*12 + as.numeric(z[2]) ,
           as.numeric(z[1]))
  })
  return(ret)
}
```

- The function `str_split()` partitions a character vector at the position of the split string and returns a **list** of substrings.
- The function `map_dbl()` from the `purrr` package[1] applies a function to each element of a vector and returns a vector of the same length and of type `double`.
- The function `ifelse` returns, if the first argument is evaluated with `TRUE`, the second argument, otherwise the third.

[1] Hadley Wickham and Garrett Grolemund. *R for Data Science - Chapter 21: Iteration*. O'Reilly, 2017.

# Functions 4/5

Test whether custom function works as intended:

```
fixheight("70")
```

```
## [1] 70
```

```
fixheight("5'10")
```

```
## [1] 70
```

```
fixheight(c("5'9","70","5'11"))
```

```
## [1] 69 70 71
```

# Functions 5/5

Apply custom function on survey heights and convert values to cm:

```
dat <- dat %>%
  mutate(height = 2.54 * fixheight(height)) %>%
  select(-numeric_height)
```

Check whether there are still observations with missing values for `height`.

```
dat %>% filter(is.na(height)) %>% select(height)
```

```
## # A tibble: 0 x 1
## # ... with 1 variable: height <dbl>
```

# Exploratory Data Analysis (EDA)

Recall:

> Imagine that we had been running a pedestrian survey in the USA. We had let pedestrians write their height and sex on a sheet of paper. Afterwards, the handwritten statements were digitalized and saved in a csv file. Our task is to summarize and describe the heights of the survey participants to a person not involved in the study.

Example question: how many females and males participated in the pedestrian survey?

```
dat %>%
  filter(sex == "Female") %>%
  nrow()
```

```
## [1] 68
```

```
dat %>%
  filter(sex == "Male") %>%
  nrow()
```

```
## [1] 79
```

```
table(dat$sex)
```

```
##
##                   Female I prefer not to disclose                     Male
##                       68                        1                       79
```

# Distributions

**Distribution**:

- basic and compact summary of a list of numbers, e.g. a variable or a column in a data frame
- example: describe list of heights to someone that has no idea what these heights are
- naive approach: list randomly selected heights

```
dat %>% select(height) %>% sample_n(10)
```

```
## # A tibble: 10 x 1
##    height
##     <dbl>
##  1   170.
##  2   178.
##  3   171.
##  4   198.
##  5   178.
##  6   175.
##  7   163.
##  8   152.
##  9   170.
## 10   178.
```

# Cumulative distribution function

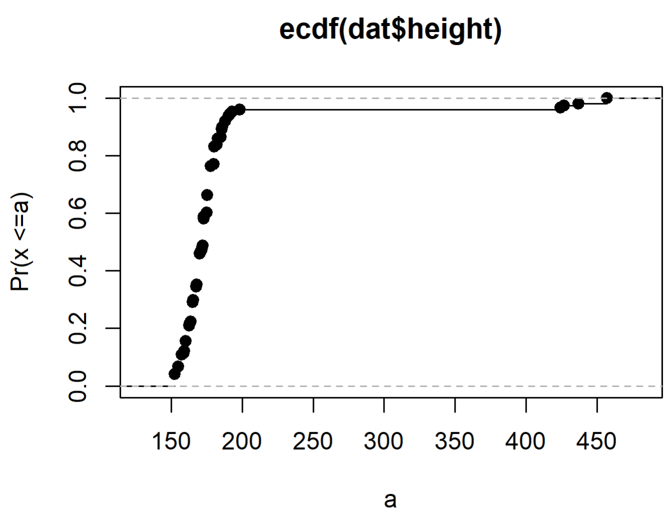
**Cumulative distribution function** (CDF): For a value $a$, what is the proportion of numbers in the list that are below $a$.

$$F(a) \equiv \Pr(x \leq a)$$

The CDF is referred to as **empirical** distribution function (ECDF) when the CDF is derived from data.

ECDF for the pedestrian heights:

```
plot(ecdf(dat$height), xlab = "a", ylab = "Pr(x ⩽ a)")
```



# Histograms

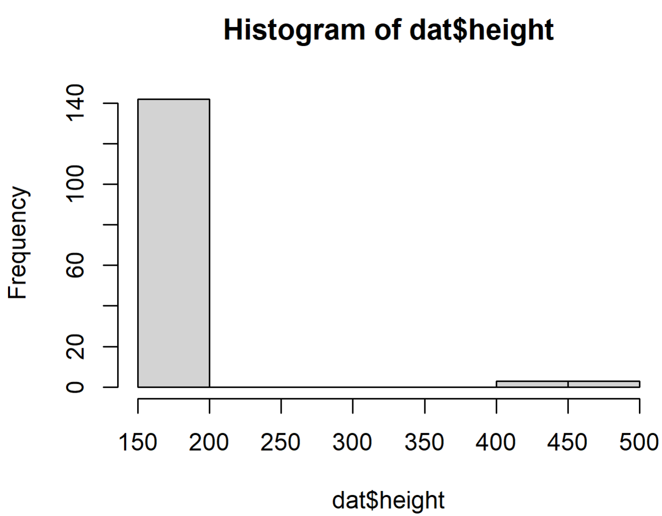
**Histograms** show the proportion of values in intervals:

$$\Pr(a \leq x \leq b) = F(b) - F(a)$$

- in practice more popular than CDFs
- easier to distinguish between different types of distributions with histograms

Histogram for the pedestrian heights:

```
hist(dat$height)
```



# Outliers 1/2

Some heights are larger than 4 m!

```r
filter(dat, height > 400) %>% select(original)
```

```
## # A tibble: 6 x 1
##   original
##   <chr>
## 1 172
## 2 180
## 3 180
## 4 180
## 5 167
## 6 168
```

# Outliers 2/2

Several heights were already specified in centimeters.

Convert values back to cm:

```
dat ← mutate(dat, height = ifelse(height > 400, height / 2.54, height))
```

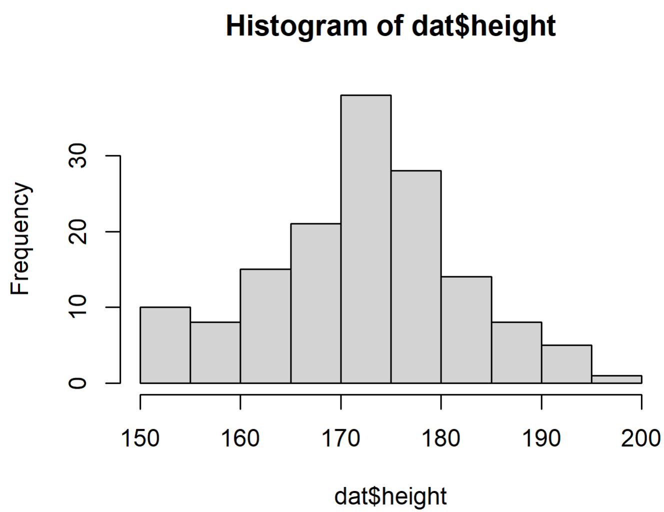
How many outliers remain?

```
filter(dat, height > 400) %>% select(original)
```

```
## # A tibble: 0 x 1
## # ... with 1 variable: original <chr>
```

# Histogram

```
hist(dat$height)
```



# Normal Distribution 1/3

The proportion of values in any given interval of the Normal distribution or Gaussian distribution can be approximated by:

$$\Pr(a < x < b) = \int_a^b \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(\frac{-(x-\mu)^2}{2\sigma^2}\right) dx$$

# Normal Distribution 2/3

Only the average $\mu$ and the standard deviation $\sigma$ are required to describe the entire population when the data is normally distributed.

$$\mu = \frac{1}{n}\sum_{i=1}^{n} x_i$$

$$Var = \sigma^2 = \frac{1}{n}\sum_{i=1}^{n}(x_i - \mu_X)^2$$

with the standard deviation being the square root of the variance.

# Normal Distribution 3/3

For the heights dataset, how many values are greater than 180 cm?

```
mean(dat$height > 180)
```

```
## [1] 0.1891892
```

```
1 - pnorm(180, mean(dat$height), sqrt(mean((dat$height-mean(dat$height))^2)))
```

```
## [1] 0.1899392
```

# Standard units 1/2

If a list of numbers follows the normal distribution, a convinent way to describe a value is the number of standard deviations away from the average. These values are in `standard units`:

```
z ← (x - mu)/sigma
```

**Question: How many standard deviations away from the average is a study participant with a height of 200 cm?**

```r
(200 - mean(dat$height)) / sqrt(mean((dat$height - mean(dat$height))^2))
```

```
## [1] 2.954377
```

# Standard units 2/2

If the original distribution is approximately normal, then these values will have a standard normal distribution: average 0 and standard deviation 1. Notice that about 95% of the values are within two standard deviation of the average:

```
pnorm(2) - pnorm(-2)
```

```
## [1] 0.9544997
```

and most values are within 3

```
pnorm(3) - pnorm(-3)
```

```
## [1] 0.9973002
```

# Boxplots 1/4

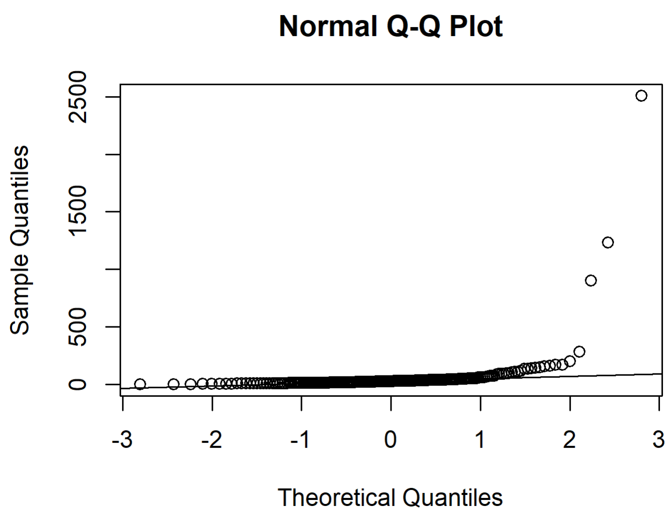
- data is not always normally distributed
  - example: *income*
- average and standard deviation are not necessarily informative since one cannot infer the distribution from just these two numbers

```
library(UsingR)
hist(exec.pay)
```

# Boxplots 2/4

Check whether the distribution of CEO compensation is normally distributed.

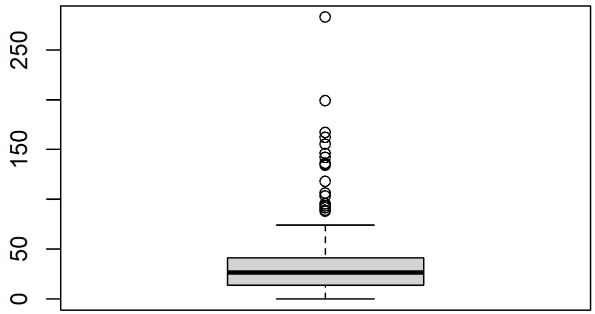
```
qqnorm(exec.pay)
qqline(exec.pay)
```



# Boxplots 3/4

Boxplot = 5 point distribution summary

```
boxplot(exec.pay[exec.pay < 500])
```




- bottom box border: 25-th percentile
- top box border: 75-th percentile
- horizontal line inside box: 50-th percentile (the median)
- bottom whisker: 25-th percentile - 1.5 * interquartile range (IQR) or minimum
- top whisker: 75-th percentile + 1.5 * interquartile range(IQR) or maximum
- extreme values are shown as single points

# Boxplots 4/4

Advantage of boxplots: easily juxtapose many distributions in one plot by showing them side by side

The distribution of men and women heights from the survey dataset:

```
boxplot(height ~ sex,
        data = dat %>% filter(sex %in% c("Female", "Male")),
        ylab = "Height of study participants in cm")
```