Cardiovascular ncRNA

Collaborative Research Centre Transregio Munich - Frankfurt (TRR 267) Non-coding RNA in the cardiovascular system Make your paper figures professionally: Scientific data analysis and visualization with R

Fatemeh Behjati Ardakani, Vangelis Theodorakis, Julien Gagneur, Marcel Schulz

10 October, 2020

Contents

Chapter 1

Foreword

The aim of these series of lectures is to provide you with all the knowledge needed in order to make state of the art plots using the R programming language and Grammar of Graphics. The lectures cover the basic concepts of R, how to manipulate data and how to generate clear and meaningful plots for your target audience, using ggplot2 an implementation of Grammar of Graphics for R.

Chapter 2

Data visualisation introduction

Data Science is crucial for many applications in a variety of fields and is the language of collaboration between biology, computer science, bioinformatics, physics, and mathematics.

Goals of this course include to introduce you to the R and Rstudio, and provide you general analytic techniques to extract knowledge, patterns, and connections between samples and variables, and how to communicate these in an intuitive and clear way. Some of the examples will be expressed in the field of bioinformatics.

2.1 What is Data Science?

Data science is an interdisciplinary field about processes and systems to extract knowledge or insights from data in various forms, either structured or unstructured. It is a continuation of some of the data analysis fields such as statistics, data mining, and predictive analytics.

The Goals of Data Science include discovering new phenomena or trends from data, enabling decisions based on facts derived from data, and communicating findings from data.

Data science at its core is the heart of any scientific method. It starts with observations, makes hypotheses, experiments, and visualizes the results in such a way that you can make claims about the validity of the hypothesis. These skills are applicable not only to whatever research topic interests you, but are also some of the hottest skills you may develop for the job market.

2.2 What you'll learn (hopefully)

Opposed to many other statistics or machine learning lectures this course is designed to teach you the practical skills you need as a data scientist. We will not focus on the complex math that went into developing the high powered statistical tools, and machine learning techniques thought in many courses. We will focus on tidy data, visualizations, and data manipulation in R.

2.3 Our tool

- R- a language and program designed for data analytics. Not great for big applications, but an environment to work with data in an easy way based on scripts.
  - Scripting: work on the fly, define few functions and data structures but use existing packages to interpret your data along with your own intuition.
  - Package developer: robust, fast code, that copes with the inherent problems of the language.

2.4 Breakdown of topics

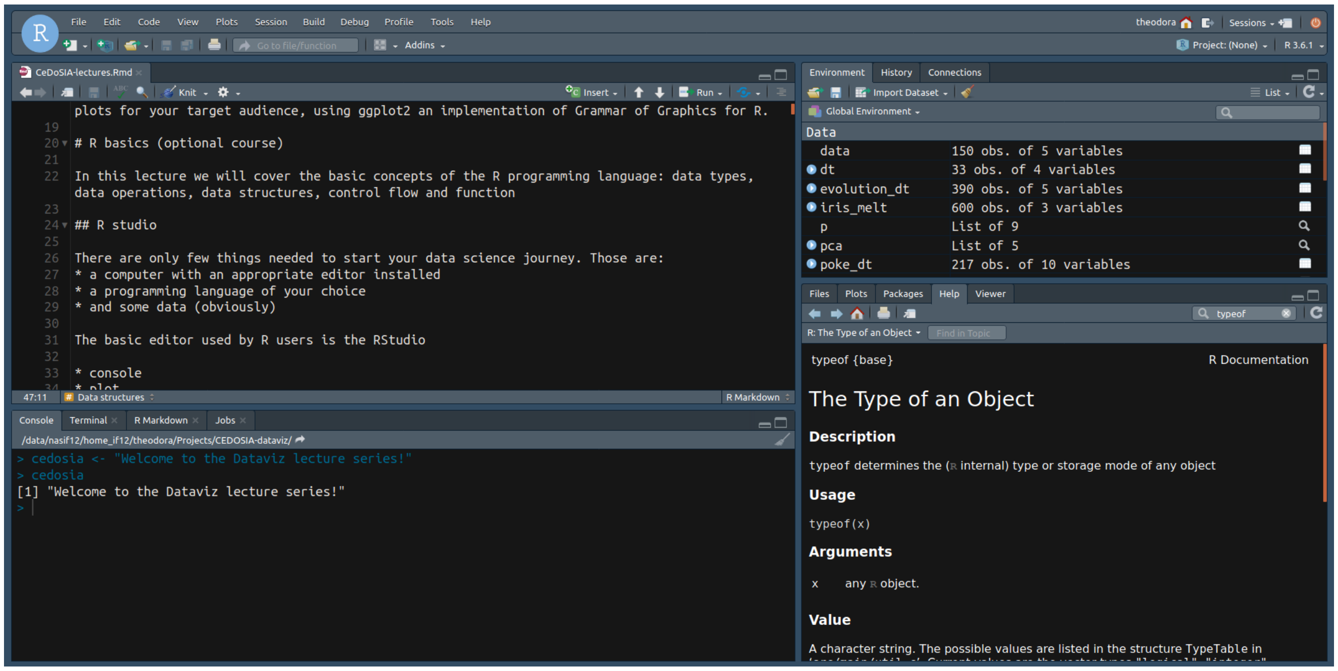
- R markdown: Notebooks that allow you to quickly publish and share your code informally so that you can see the results and the code that generated it. (This is an R markdown file)
- Tidy data: standard way of structuring data for efficient memory storage and efficient operations
- Visualization with ggplot2: flexible grammar and frame-work to visualize data in concise and easy ways.

Chapter 3

Introduction to R and Rstudio

- R is the language of this course. It shares some aspects with python, but the syntax is often unique compared to other languages. All R files are named with a .R extension (dot - R)

- Rstudio is an editor, it succinctly organizes your session into 4 windows each set up for you to do certain tasks in each window. Edit and write code (editor), run and execute code (console), list and name objects, and a window to show output in either function help, and plots.



```
plots for your target audience, using ggplot2 an implementation of Grammar of Graphics for R.

# R basics (optional course)

In this lecture we will cover the basic concepts of the R programming language: data types,
data operations, data structures, control flow and function

## R studio

There are only few things needed to start your data science journey. Those are:
* a computer with an appropriate editor installed
* a programming language of your choice
* and some data (obviously)

The basic editor used by R users is the RStudio

* console
* plot
```

```
> cedosia <- "Welcome to the Dataviz lecture series!"
> cedosia
[1] "Welcome to the Dataviz lecture series!"
>
```

```
Data
  data           150 obs. of 5 variables
● dt             33 obs. of 4 variables
● evolution_dt   390 obs. of 5 variables
● iris_melt      600 obs. of 3 variables
  p              List of 9
● pca            List of 5
● poke_dt        217 obs. of 10 variables
```

typeof {base} R Documentation

The Type of an Object

Description

typeof determines the (R internal) type or storage mode of any object

Usage

typeof(x)

Arguments

x any R object.

Value

A character string. The possible values are listed in the structure TypeTable in

Figure 3.1: Rstudio

The most commonly used editor by R users is the RStudio. It consists of the following sections such as:

- the main script section, for writing scripts [top left section] (Ctrl+1 to focus)
- the console tab, for typing R commands directly [bottom left section as tab] (Ctrl+2 to focus)
- the terminal tab, for direct access to your system shell [bottom left section] (Shift+Alt+T to focus)
- the plot tab, where you see the last plot generated [bottom right section as tab]
- the help tab, with useful documentation of R functions [bottom right section as tab] (F1 on the name of a function or Ctrl+3)
- the history tab, with the list of the R commands used [top right section as tab]
- the environment tab, with the created variables and functions loaded [top right section as tab]

- and the packages tab, with the available/loaded R packages [bottom right section as tab]

Check the View menu to find out the rest of useful shortcuts!

3.1 Introduction to basic R syntax

- All code is run in the console (bottom left) you can write and alter code directly in here or in the editor above (top left) for easier editing/debugging
- All of your questions and issues can be investigated Here
- R examples:
  - basic operators

```
x <- 10
x
```

```
## [1] 10
```

```
x + 5
```

```
## [1] 15
```

```
y <- x + 5
y
```

```
## [1] 15
```

```
x == y
```

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

```
x
```

```
## [1] 10
```

  - built-in methods

```
x <- rnorm(10)  #returns a vector of random numbers based on the normal distribution
mean(x)
```

```
## [1] 0.3654494
```

- **?** operator
  `?mean` shows you information about the function you are calling. Look to the bottom right window.

We will introduce further details of the R language as they are needed throughout the course and refer you to the Appendix II for a more detailed introduction to R.

Chapter 4

Data Wrangling using Data.table

Data Wrangling includes tasks of processing raw data into interesting formats. . . .

Data.tables are a modern implementation of tables in R. We will exclusively used data.tables in this course. Base R provides a similar structure called data.frames. However, those are a lot slower and often a little more complicated to use, so we highly recommend using data.table.

Similar to a data.frame, but because it modifies columns by *reference*, a data.table is a memory efficient and faster implementation of data.frame offering:

- sub-setting
- ordering
- merging

It accepts all data.frame functions, it has a shorter and more flexible syntax (not so straightforward in the beginning but pays off) and saves time on two fronts:

- programming (easier to code, read, debug and maintain)
- computing (fast and memory efficient)

The general form of data.table syntax is:

```
DT[ i,  j,  by ] # + extra arguments
    |   |   |
    |   |    -------> grouped by what?
    |    -------> what to do?
     ---> on which rows?
```

The way to read this out loud is: "Take DT, subset rows by i, then compute j grouped by by. Here are some basic usage examples expanding on this definition.

Like data.frame, data.table is a list of vectors. It doesn't have row names; instead, it's a collection of attributes.

Each column can be a different type, including a list

It has enhanced functionality in `[`. We can make operations inside `[]`.

For a detailed documentation go check the help page for data.table!

4.1 Creating and loading data

To create a data.table, we just name its columns and populate them. All the columns have to have the same length.

```
# install.packages('data.table')
library(data.table)
DT <- data.table(x = rep(c("a", "b", "c"), each = 3), y = c(1, 3, 6), v = 1:9)
DT  # note how column y was recycled
```

```
##    x y v
## 1: a 1 1
## 2: a 3 2
## 3: a 6 3
## 4: b 1 4
## 5: b 3 5
## 6: b 6 6
## 7: c 1 7
## 8: c 3 8
## 9: c 6 9
```

If we want to convert any other R object to a data.table, all we have to do is to call the `as.data.table()` function.

```
# This way we can for example convert any inbuilt data set into a data.table:
titanic_dt <- as.data.table(Titanic)
class(titanic_dt)
```

```
## [1] "data.table" "data.frame"
```

Here you can see that the class function informs us that DT is both a data.table and a data.frame as data.tables inherit from data.frames, an older table class in R.

Alternativly we can read files from disk and process them using data.table. The easiest way to do so is to use the function `fread()`.

```
flights <- fread("path_to_file/flightsLAX.csv")
```

Calling `head()` on the table showes us the first few lines of the table and we observe that reading the file was succesfull.

```
head(flights)
```

```
##    YEAR MONTH DAY DAY_OF_WEEK AIRLINE FLIGHT_NUMBER TAIL_NUMBER ORIGIN_AIRPORT
## 1: 2015     1   1           4      AA          2336      N3KUAA            LAX
## 2: 2015     1   1           4      AA           258      N3HYAA            LAX
## 3: 2015     1   1           4      US          2013      N584UW            LAX
## 4: 2015     1   1           4      DL          1434      N547US            LAX
## 5: 2015     1   1           4      AA           115      N3CTAA            LAX
## 6: 2015     1   1           4      UA          1545      N76517            LAX
##    DESTINATION_AIRPORT DEPARTURE_TIME AIR_TIME DISTANCE ARRIVAL_TIME
## 1:                 PBI              2      263     2330          741
## 2:                 MIA             15      258     2342          756
## 3:                 CLT             44      228     2125          753
## 4:                 MSP             35      188     1535          605
## 5:                 MIA            103      255     2342          839
## 6:                 IAH            112      156     1379          607
```

4.2 Inspecting tables

A first step in any analysis should involve inspecting the data we just read in. This often starts by looking the head of the table as we did above.

The next information we are often interested in is the size of our data set. We can use the following commands to obtain it.

```
nrow(flights)  # ncol(flights)
```

```
## [1] 389369
```

```
dim(flights)
```

```
## [1] 389369     13
```

4.2.1 Technical detail

The structure of the data.table essentially is a list of list. To get the dimensions of our data.table we can either use the `dim()` function to get the dimensions of our data.table (M rows x N columns) or to use the `ncol()` `nrow()` functions.

```
class(DT)
```

```
## [1] "data.table" "data.frame"
```

```
is.list(DT)
```

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

```
nrow(DT)  # ncol(DT)
```

```
## [1] 9
```

```
dim(DT)
```

```
## [1] 9 3
```

Next we are often interested in the range or the number of occurence of values in certain columns. To obtain this information we can use the summary command on the table.

```
summary(flights[, 1:6])
```

```
##       YEAR          MONTH              DAY         DAY_OF_WEEK
##  Min.   :2015   Min.   : 1.000   Min.   : 1.0   Min.   :1.000
##  1st Qu.:2015   1st Qu.: 3.000   1st Qu.: 8.0   1st Qu.:2.000
##  Median :2015   Median : 6.000   Median :16.0   Median :4.000
##  Mean   :2015   Mean   : 6.198   Mean   :15.7   Mean   :3.934
##  3rd Qu.:2015   3rd Qu.: 9.000   3rd Qu.:23.0   3rd Qu.:6.000
##  Max.   :2015   Max.   :12.000   Max.   :31.0   Max.   :7.000
##    AIRLINE          FLIGHT_NUMBER
##  Length:389369      Min.   :   1
##  Class :character   1st Qu.: 501
##  Mode  :character   Median :1296
##                     Mean   :1905
##                     3rd Qu.:2617
##                     Max.   :6896
```

This provides us already a lot of information about our data. We can for example see that all data is from 2015 as all values in the YEAR column are 2015. But for categorical data this is not very insight full, as we can see for the AIRLINE column.

To investigate categorical columns we can have a look at their unique elements using:

```
flights[, unique(AIRLINE)]
```

```
##  [1] "AA" "US" "DL" "UA" "OO" "AS" "B6" "NK" "VX" "WN" "HA" "F9" "MQ"
```

This command provided us the airline identifiers present in the dataset. Another valueable information for categoricals is how often each category occurs. This can be obtained using the following commands:

```
flights[, table(AIRLINE)]
```

```
## AIRLINE
##    AA    AS    B6    DL    F9    HA    MQ    NK    OO    UA    US    VX    WN
## 65483 16144  8216 50343  2770  3112   368  8688 73389 54862  7374 23598 75022
```

Or:

```
flights[, .N, by = "AIRLINE"]
```

```
##     AIRLINE     N
##  1:      AA 65483
##  2:      US  7374
##  3:      DL 50343
##  4:      UA 54862
##  5:      OO 73389
##  6:      AS 16144
##  7:      B6  8216
##  8:      NK  8688
##  9:      VX 23598
## 10:      WN 75022
## 11:      HA  3112
## 12:      F9  2770
## 13:      MQ   368
```

Especially the second command will be explained in detail later.

4.3 Accessing rows and columns of data.tables

Data.tables can be subsetted by indices, if we for example care about the second element of the table. We can do the following.

```
flights[2, ]  # Access the 2nd row (also flights[2] or flights[i = 2])
```

```
##    YEAR MONTH DAY DAY_OF_WEEK AIRLINE FLIGHT_NUMBER TAIL_NUMBER ORIGIN_AIRPORT
## 1: 2015     1   1           4      AA           258      N3HYAA            LAX
##    DESTINATION_AIRPORT DEPARTURE_TIME AIR_TIME DISTANCE ARRIVAL_TIME
## 1:                 MIA             15      258     2342          756
```

```
flights[1:2]  # Access multiple consecutive rows.
```

```
##    YEAR MONTH DAY DAY_OF_WEEK AIRLINE FLIGHT_NUMBER TAIL_NUMBER ORIGIN_AIRPORT
## 1: 2015     1   1           4      AA          2336      N3KUAA            LAX
## 2: 2015     1   1           4      AA           258      N3HYAA            LAX
##    DESTINATION_AIRPORT DEPARTURE_TIME AIR_TIME DISTANCE ARRIVAL_TIME
## 1:                 PBI              2      263     2330          741
## 2:                 MIA             15      258     2342          756
```

```
flights[c(3, 5)]  # Access multiple rows.
```

```
##    YEAR MONTH DAY DAY_OF_WEEK AIRLINE FLIGHT_NUMBER TAIL_NUMBER ORIGIN_AIRPORT
## 1: 2015     1   1           4      US          2013      N584UW            LAX
## 2: 2015     1   1           4      AA           115      N3CTAA            LAX
##    DESTINATION_AIRPORT DEPARTURE_TIME AIR_TIME DISTANCE ARRIVAL_TIME
## 1:                 CLT             44      228     2125          753
## 2:                 MIA            103      255     2342          839
```

Additionally we can do the same for columns:

```
head(flights[, 2])
```

```
##    MONTH
## 1:     1
## 2:     1
## 3:     1
## 4:     1
## 5:     1
## 6:     1
```

```
head(flights[, c(3, 5)])
```

```
##    DAY AIRLINE
## 1:   1      AA
## 2:   1      AA
## 3:   1      US
## 4:   1      DL
## 5:   1      AA
## 6:   1      UA
```

Alternatively to subsetting by indices we can subset columns by their names.

```
head(flights[, .(MONTH)])
```

```
##    MONTH
## 1:     1
## 2:     1
## 3:     1
## 4:     1
## 5:     1
## 6:     1
```

```
head(flights[, .(DAY, AIRLINE)])
```

```
##    DAY AIRLINE
## 1:   1      AA
## 2:   1      AA
## 3:   1      US
## 4:   1      DL
## 5:   1      AA
## 6:   1      UA
```

4.4 Sub-setting rows according to some condition

A often more usefull way is to subset rows using conditions! We can subset a data table using the following operators:

- `==`
- `<`
- `>`
- `!=`
- `%in%`

For example if we are interested in analysing all the flights operated by “AA” (American Airlines) we can do that using the following command:

```
head(flights[AIRLINE == "AA"])
```

```
##    YEAR MONTH DAY DAY_OF_WEEK AIRLINE FLIGHT_NUMBER TAIL_NUMBER ORIGIN_AIRPORT
## 1: 2015     1   1           4      AA          2336      N3KUAA            LAX
## 2: 2015     1   1           4      AA           258      N3HYAA            LAX
## 3: 2015     1   1           4      AA           115      N3CTAA            LAX
## 4: 2015     1   1           4      AA          2410      N3BAAA            LAX
## 5: 2015     1   1           4      AA          1515      N3HMAA            LAX
## 6: 2015     1   1           4      AA          1686      N4XXAA            LAX
##    DESTINATION_AIRPORT DEPARTURE_TIME AIR_TIME DISTANCE ARRIVAL_TIME
## 1:                 PBI              2      263     2330          741
## 2:                 MIA             15      258     2342          756
## 3:                 MIA            103      255     2342          839
## 4:                 DFW            600      150     1235         1052
## 5:                 ORD            557      202     1744         1139
## 6:                 STL            609      183     1592         1134
```

Alternatively if we are interested in all flights from any destination to the airports in NYC (JFK and LGA), we can subset the rows using the following command:

```
flights[DESTINATION_AIRPORT %in% c("LGA", "JFK")]
```

```
##        YEAR MONTH DAY DAY_OF_WEEK AIRLINE FLIGHT_NUMBER TAIL_NUMBER
##     1: 2015     1   1           4      B6            24      N923JB
##     2: 2015     1   1           4      DL           476      N196DN
##     3: 2015     1   1           4      AA           118      N788AA
##     4: 2015     1   1           4      VX           404      N621VA
##     5: 2015     1   1           4      UA           275      N598UA
##    ---
## 12011: 2015    12  31           4      AA           180      N796AA
## 12012: 2015    12  31           4      B6           524      N934JB
## 12013: 2015    12  31           4      B6           624      N942JB
## 12014: 2015    12  31           4      DL          1262      N394DL
## 12015: 2015    12  31           4      B6          1124      N943JB
##        ORIGIN_AIRPORT DESTINATION_AIRPORT DEPARTURE_TIME AIR_TIME DISTANCE
##     1:            LAX                 JFK            620      279     2475
##     2:            LAX                 JFK            650      274     2475
##     3:            LAX                 JFK            650      268     2475
##     4:            LAX                 JFK            728      268     2475
##     5:            LAX                 JFK            806      277     2475
##    ---
## 12011:            LAX                 JFK           1640      259     2475
## 12012:            LAX                 JFK           1645      261     2475
## 12013:            LAX                 JFK           2107      280     2475
## 12014:            LAX                 JFK           2244      256     2475
## 12015:            LAX                 JFK           2349      274     2475
##        ARRIVAL_TIME
##     1:         1413
##     2:         1458
##     3:         1436
##     4:         1512
##     5:         1606
##    ---
## 12011:           18
## 12012:           18
## 12013:          513
## 12014:          625
## 12015:          748
```

Additionally we can concatenate multiple condition using the logical OR `|` or AND `&` operator. If we for example want to inspect all flights departing between 6am and 7am operated by American Airlines we can use the following statement:

```
head(flights[AIRLINE == "AA" & DEPARTURE_TIME > 600 & DEPARTURE_TIME < 700])
```

```
##    YEAR MONTH DAY DAY_OF_WEEK AIRLINE FLIGHT_NUMBER TAIL_NUMBER ORIGIN_AIRPORT
## 1: 2015     1   1           4      AA          1686      N4XXAA            LAX
## 2: 2015     1   1           4      AA          1361      N3KYAA            BNA
## 3: 2015     1   1           4      AA          2420      N3ERAA            LAX
## 4: 2015     1   1           4      AA           338      N3DJAA            LAX
## 5: 2015     1   1           4      AA          2424      N3DLAA            LAX
## 6: 2015     1   1           4      AA           222      N3JSAA            LAX
##    DESTINATION_AIRPORT DEPARTURE_TIME AIR_TIME DISTANCE ARRIVAL_TIME
## 1:                 STL            609      183     1592         1134
## 2:                 LAX            607      255     1797          847
## 3:                 DFW            619      149     1235         1119
## 4:                 SFO            644       55      337          803
## 5:                 DFW            641      146     1235         1149
## 6:                 BOS            650      271     2611         1442
```

4.5 Accessing a data.table by columns (DT[i, j, by])

Once you're inside the `[]`, you're in the data.table environment. Inside this scope, there's no need to put the column names in quotation marks, as columns are seen as variables already.

It is not advisable to access a column by its number. **Use the column name instead**.

The table format can change (new columns can be added), so using column names prevents bugs e.g. if you have a data set with 50 columns, how do you know which one is column 18?

This way the code is more readable: `flights[, TAIL_NUMBER]` instead of `flights[, 7]`.

```
flights[1:10, TAIL_NUMBER]  # Access column x (also DT$x or DT[j=x]).
```

```
##  [1] "N3KUAA" "N3HYAA" "N584UW" "N547US" "N3CTAA" "N76517" "N925SW" "N719SK"
##  [9] "N435SW" "N560SW"
```

```
flights[4, TAIL_NUMBER]  # Access a specific cell.
```

```
## [1] "N547US"
```

When accessing many columns, we probably want to return a data.table instead of a vector. For that, we need to provide R with a list, so we use `list(colA, colB)`, or its simplified version `.(colA, colB)`.

```
# Note that 1 and 3 were coerced into strings because a vector can have only 1
# type
flights[1:2, c(TAIL_NUMBER, ORIGIN_AIRPORT)]
```

```
## [1] "N3KUAA" "N3HYAA" "LAX"    "LAX"
```

```
# Access a specific subset. Data.frame: DF[1:2, c('x','y')]
flights[1:2, list(TAIL_NUMBER, ORIGIN_AIRPORT)]
```

```
##    TAIL_NUMBER ORIGIN_AIRPORT
## 1:      N3KUAA            LAX
## 2:      N3HYAA            LAX
```

```
# Same as before.
flights[1:2, .(TAIL_NUMBER, ORIGIN_AIRPORT)]
```

```
##    TAIL_NUMBER ORIGIN_AIRPORT
## 1:      N3KUAA            LAX
## 2:      N3HYAA            LAX
```

4.5.1 Technical Detail Data.table environment

The following examples are to show how the `[]` bring us inside the data.table environment.

Using DT and DF, compute the product of columns *y* and *v*.

Use the `with(data, expr, ...)` function, which evaluates an R expression in an environment constructed from `data`.

```r
# We enter the environment of DT, and simply compute the product
with(DT, y * v)

# Easier way, with [] we're inside the environment. Data.table runs a
# with(dt,...) on the j argument
DT[, y * v]

# In data.frame, the [] doesn't imply we're inside the environment
DF[, y * v]
```

4.6 Basic operations

We saw already that inside the `[]`, columns are seen as variables, so we can apply functions to them.

```r
# Similar to mean(flights[, AIR_TIME])
flights[, mean(AIR_TIME, na.rm = TRUE)]
```

```
## [1] 162.1379
```

```r
flights[AIRLINE == "OO", mean(AIR_TIME, na.rm = TRUE)]
```

```
## [1] 68.02261
```

To compute operations in multiple columns, we must provide a list (unless we want the result to be a vector).

```r
# Same as flights[, .(mean(AIR_TIME), median(AIR_TIME))]
flights[, list(mean(AIR_TIME, na.rm = TRUE), median(AIR_TIME, na.rm = TRUE))]
```

```
##          V1  V2
## 1: 162.1379 150
```

```r
# Give meaningful names
flights[, .(mean_AIR_TIME = mean(AIR_TIME, na.rm = TRUE), median_AIR_TIME = median(AIR_TIME,
    na.rm = TRUE))]
```

```
##    mean_AIR_TIME median_AIR_TIME
## 1:      162.1379             150
```

4.6.1 Technical detail We can also apply basic operations on multiple columns

`sapply(DT, FUN)`, applies function FUN column-wise to DT. Remember that `sapply` returns a vector, while `lapply` returns a list.

```r
sapply(iris_dt, class)  # Try the same with lapply
```

```
## Error in lapply(X = X, FUN = FUN, ...): object 'iris_dt' not found
```

```
sapply(iris_dt, sum)
```

```
## Error in lapply(X = X, FUN = FUN, ...): object 'iris_dt' not found
```

```
# Note that we can access columns stored as variables by setting with=F.  In this
# case, `colnames(iris_dt)!='Species'` returns a logical vector and ` iris_dt` is
# subseted by the logical vector

# Same as sapply(iris_dt[, 1:4], sum) sapply(iris_dt[,
# colnames(iris_dt)!='Species', with = F], sum)
```

4.7 The ‘by’ option (DT[i, j, by])

The `by =` option allows us to apply a function to groups wtihin a data.table. For example, we can use the `by =` to compute the mean flight time per airline:

```
# Compute the mean air time of every airline
flights[, .(mean_AIRTIME = mean(AIR_TIME, na.rm = TRUE)), by = AIRLINE]
```

```
##     AIRLINE mean_AIRTIME
##  1:      AA    219.48133
##  2:      US    210.39488
##  3:      DL    207.07201
##  4:      UA    211.62008
##  5:      OO     68.02261
##  6:      AS    141.01870
##  7:      B6    309.79568
##  8:      NK    179.55828
##  9:      VX    185.36374
## 10:      WN    105.19976
## 11:      HA    307.95961
## 12:      F9    159.94041
## 13:      MQ    102.15210
```

This way we can easily spot that one airline conducts on average shorter flights.

```
# Compute the mean and standard deviation of the air time of every airline
flights[, .(mean_AIRTIME = mean(AIR_TIME, na.rm = TRUE), sd_AIR_TIME = sd(AIR_TIME,
    na.rm = TRUE)), by = AIRLINE]
```

```
##     AIRLINE mean_AIRTIME sd_AIR_TIME
##  1:      AA    219.48133   92.889719
##  2:      US    210.39488  105.224833
##  3:      DL    207.07201   88.908566
##  4:      UA    211.62008   94.832456
##  5:      OO     68.02261   41.065036
##  6:      AS    141.01870   51.806424
##  7:      B6    309.79568   28.457740
##  8:      NK    179.55828   78.194706
##  9:      VX    185.36374  113.504572
## 10:      WN    105.19976   69.257334
## 11:      HA    307.95961   23.905491
## 12:      F9    159.94041   61.412379
## 13:      MQ    102.15210    8.531046
```

Although we could write `flights[i = 5, j = AIRLINE]`, we usually ommit the `i =` and `j =` from the syntax, and write `flights[5, ARILINE]` instead. However, for clarity we usually include the `by =` in the syntax.

4.8 Counting occurences (the `.N` command)

The `.N` is a special in-built variable that counts the number observations within a table.

Evaluating `.N` alone is equal to `nrow()` of a table.

```
flights[, .N]
```

```
## [1] 389369
```

```
nrow(flights)
```

```
## [1] 389369
```

But the `.N` command becomes a lot more powerful when used with grouping or conditioning. We already saw earlier how we can use it to count the number of occurrences of elements in categorical columns.

```
# Get the number of flights for each AIRLINE
flights[, .N, by = "AIRLINE"]
```

```
##     AIRLINE     N
##  1:      AA 65483
##  2:      US  7374
##  3:      DL 50343
##  4:      UA 54862
##  5:      OO 73389
##  6:      AS 16144
##  7:      B6  8216
##  8:      NK  8688
##  9:      VX 23598
## 10:      WN 75022
## 11:      HA  3112
## 12:      F9  2770
## 13:      MQ   368
```

Remembering the data.table definition: “Take **DT**, subset rows using **i**, then select or calculate **j**, grouped by **by**”, we can build even more powerful statements using all three elements.

```
# For each airline, get the number of observations to NYC (JFK)
flights[DESTINATION_AIRPORT == "JFK", .N, by = "AIRLINE"]
```

```
##    AIRLINE    N
## 1:      B6 2488
## 2:      DL 2546
## 3:      AA 3804
## 4:      VX 1652
## 5:      UA 1525
```

4.9 Creating new columns (the := command)

The `:=` operator updates the data.table you are working on, so writing DT <- DT[,... := ...] is redundant. This operator, plus all `set` functions (TODO clarify), change their input by *reference*. No copy of the object is made, that is why it is faster and uses less memory.

```
# Add a new column called SPEED whose value is the DISTANCE divided by AIR_TIME
flights[, `:=`(SPEED, DISTANCE/AIR_TIME * 60)]
head(flights)
```

```
##    YEAR MONTH DAY DAY_OF_WEEK AIRLINE FLIGHT_NUMBER TAIL_NUMBER ORIGIN_AIRPORT
## 1: 2015     1   1           4      AA          2336      N3KUAA            LAX
## 2: 2015     1   1           4      AA           258      N3HYAA            LAX
## 3: 2015     1   1           4      US          2013      N584UW            LAX
## 4: 2015     1   1           4      DL          1434      N547US            LAX
## 5: 2015     1   1           4      AA           115      N3CTAA            LAX
## 6: 2015     1   1           4      UA          1545      N76517            LAX
##    DESTINATION_AIRPORT DEPARTURE_TIME AIR_TIME DISTANCE ARRIVAL_TIME    SPEED
## 1:                 PBI              2      263     2330          741 531.5589
## 2:                 MIA             15      258     2342          756 544.6512
## 3:                 CLT             44      228     2125          753 559.2105
## 4:                 MSP             35      188     1535          605 489.8936
## 5:                 MIA            103      255     2342          839 551.0588
## 6:                 IAH            112      156     1379          607 530.3846
```

Having computed a new column using the := operator we can use it for further analyses.

```
flights[, .(mean_AIR_TIME = mean(AIR_TIME, na.rm = TRUE), mean_SPEED = mean(SPEED,
    na.rm = TRUE), mean_DISTANCE = mean(DISTANCE, na.rm = TRUE)), by = AIRLINE]
```

```
##     AIRLINE mean_AIR_TIME mean_SPEED mean_DISTANCE
##  1:      AA     219.48133   461.2839     1739.2331
##  2:      US     210.39488   452.1641     1658.2581
##  3:      DL     207.07201   466.0330     1656.2165
##  4:      UA     211.62008   464.2928     1693.5504
##  5:      OO      68.02261   349.5549      437.2337
##  6:      AS     141.01870   439.0120     1040.0340
##  7:      B6     309.79568   484.8242     2486.1489
##  8:      NK     179.55828   450.0221     1402.1591
##  9:      VX     185.36374   433.0870     1432.5384
## 10:      WN     105.19976   409.3803      760.2593
## 11:      HA     307.95961   497.3118     2537.8107
## 12:      F9     159.94041   461.0684     1235.6664
## 13:      MQ     102.15210   435.5580      737.0000
```

Now we can see that the flights by the carrier “OO” are not just shorter, but also slow. This could for example lead us to the hypothesis, that “OO” is a small regional carrier, which operates slower planes.

Additionally we can use the := operator to remove columns. If we for example observe that tail numbers are not important for our analysis we can remove them with the following statement:

```
flights[, `:=`(TAIL_NUMBER, NULL)]
head(flights)
```

```
##    YEAR MONTH DAY DAY_OF_WEEK AIRLINE FLIGHT_NUMBER ORIGIN_AIRPORT
## 1: 2015     1   1           4      AA          2336            LAX
## 2: 2015     1   1           4      AA           258            LAX
## 3: 2015     1   1           4      US          2013            LAX
## 4: 2015     1   1           4      DL          1434            LAX
## 5: 2015     1   1           4      AA           115            LAX
## 6: 2015     1   1           4      UA          1545            LAX
##    DESTINATION_AIRPORT DEPARTURE_TIME AIR_TIME DISTANCE ARRIVAL_TIME    SPEED
## 1:                 PBI              2      263     2330          741 531.5589
## 2:                 MIA             15      258     2342          756 544.6512
## 3:                 CLT             44      228     2125          753 559.2105
## 4:                 MSP             35      188     1535          605 489.8936
## 5:                 MIA            103      255     2342          839 551.0588
## 6:                 IAH            112      156     1379          607 530.3846
```

Here we observe, that the tail numbers are gone from the data.table

4.9.1 Technical Detail Multiple assignments

With the following syntax we can assign multiple new columns at once.

```
# Add columns with sepal and petal area. Note the syntax of multiple assignment.
# iris_dt[, `:=` (Sepal.Area = Sepal.Length * Sepal.Width, Petal.Area =
# Petal.Length * Petal.Width)][1:3]
```

You can also delete columns by using the := command.

```
# Let's assume setosa flowers are orange, versicolor purple and virginica pink.
# Add a column with these colors. iris_dt[Species == 'setosa', color := 'orange']
# iris_dt[Species == 'versicolor', color := 'purple'] iris_dt[Species ==
# 'virginica', color := 'pink'] unique(iris_dt[, .(Species, color)])

# We can delete this new column by setting it to NULL iris_dt[, color := NULL]
# colnames(iris_dt)
```

4.10 By reference

What do we mean when we say that data.table modifies columns *by reference*? It means that no new copy of the object is made in the memory, unless we actually create one using `copy()`.

```
or_dt <- data.table(a = 1:10, b = 11:20)
# No new object is created, both new_dt and or_dt point to the same memory chunk.
new_dt <- or_dt
new_dt[, `:=`(ab, a * b)]
colnames(or_dt)  # or_dt was also affected by changes in new_dt
```

```
## [1] "a"  "b"  "ab"
```

```
or_dt <- data.table(a = 1:10, b = 11:20)
copy_dt <- copy(or_dt)  # By creating a copy, we have 2 objects in memory
copy_dt[, `:=`(ab, a * b)]
colnames(or_dt)  # Changes in the copy don't affect the original
```

```
## [1] "a" "b"
```

4.11 Merging

Merge, join or combine 2 data tables into one by common column(s).

```
merge(x, y, by, by.x, by.y, all, all.x, all.y, ...)
```

In data.table, the default is to merge by shared *key* columns while in data.frame, the default is to merge by columns with the same name. We can also specify the columns to merge `by`.

There are different types of merging:

- **Inner (default)**: consider only rows with matching values in the key columns.
- **Outer or full**: return all rows and columns from x and y. If there are no matching values, return NAs.
- **Left (all.x)**: consider all rows from x, even if they have no matching row in y.
- **Right (all.y)**: consider all rows from y, even if they have no matching row in x.

4.11.1 Merging - example

If we for example want to know how the airports are called behind the IATA_CODE codes, we can load another tabe containing additional information about the airports.

```
airports <- fread("path_to_file/airports.csv")
```

```
head(airports)
```

```
##    IATA_CODE                             AIRPORT        CITY STATE COUNTRY
## 1:       ABE Lehigh Valley International Airport   Allentown    PA     USA
## 2:       ABI            Abilene Regional Airport     Abilene    TX     USA
## 3:       ABQ   Albuquerque International Sunport Albuquerque    NM     USA
## 4:       ABR           Aberdeen Regional Airport    Aberdeen    SD     USA
## 5:       ABY  Southwest Georgia Regional Airport      Albany    GA     USA
## 6:       ACK         Nantucket Memorial Airport   Nantucket    MA     USA
##    LATITUDE  LONGITUDE
## 1: 40.65236  -75.44040
## 2: 32.41132  -99.68190
## 3: 35.04022 -106.60919
## 4: 45.44906  -98.42183
## 5: 31.53552  -84.19447
## 6: 41.25305  -70.06018
```

```
head(merge(flights, airports, by.x = "DESTINATION_AIRPORT", by.y = "IATA_CODE"))
```

```
##    DESTINATION_AIRPORT YEAR MONTH DAY DAY_OF_WEEK AIRLINE FLIGHT_NUMBER
## 1:                 ABQ 2015     1   1           4      OO          2575
## 2:                 ABQ 2015     1   1           4      WN          2077
## 3:                 ABQ 2015     1   1           4      OO          2623
## 4:                 ABQ 2015     1   1           4      WN          2520
## 5:                 ABQ 2015     1   1           4      OO          6474
## 6:                 ABQ 2015     1   1           4      WN          4721
##    ORIGIN_AIRPORT DEPARTURE_TIME AIR_TIME DISTANCE ARRIVAL_TIME    SPEED
## 1:            LAX            921       87      677         1214 466.8966
## 2:            LAX            943       88      677         1232 461.5909
## 3:            LAX           1449       89      677         1750 456.4045
## 4:            LAX           1851       90      677         2134 451.3333
## 5:            LAX           1951      100      677         2256 406.2000
## 6:            LAX           2041       88      677         2328 461.5909
##                              AIRPORT        CITY STATE COUNTRY LATITUDE
## 1: Albuquerque International Sunport Albuquerque    NM     USA 35.04022
## 2: Albuquerque International Sunport Albuquerque    NM     USA 35.04022
## 3: Albuquerque International Sunport Albuquerque    NM     USA 35.04022
## 4: Albuquerque International Sunport Albuquerque    NM     USA 35.04022
## 5: Albuquerque International Sunport Albuquerque    NM     USA 35.04022
## 6: Albuquerque International Sunport Albuquerque    NM     USA 35.04022
##    LONGITUDE
## 1: -106.6092
## 2: -106.6092
## 3: -106.6092
## 4: -106.6092
## 5: -106.6092
## 6: -106.6092
```

4.11.2 Merging - inner and outer examples

Below we created some artificial tables to showcase the differences between the different types of merges.

```
dt1 <- data.table(table = "table1", id = c(1, 2, 3), a = rnorm(3))
dt2 <- data.table(table = "table2", id = c(1, 2, 4), b = rnorm(3))
```

```
# Inner merge: default one, all = FALSE
merge(dt1, dt2, by = "id", all = F)
```

```
##    id table.x          a table.y         b
## 1:  1  table1 -0.5345377  table2 1.0071995
## 2:  2  table1 -0.6252274  table2 0.7192918
```

```
# Outer (full) merge: all = TRUE
merge(dt1, dt2, by = "id", all = T)
```

```
##    id table.x          a table.y          b
## 1:  1  table1 -0.5345377  table2  1.0071995
## 2:  2  table1 -0.6252274  table2  0.7192918
## 3:  3  table1  0.9138487    <NA>         NA
## 4:  4    <NA>         NA  table2 -0.6047117
```

Note that the column order got changed after the merging.

```
# Left merge: all.x = TRUE
merge(dt1, dt2, by = "id", all.x = T)[]
```

```
##    id table.x          a table.y         b
## 1:  1  table1 -0.5345377  table2 1.0071995
## 2:  2  table1 -0.6252274  table2 0.7192918
## 3:  3  table1  0.9138487    <NA>        NA
```

```
# Right merge: all.y = TRUE
merge(dt1, dt2, by = "id", all.y = T)[]
```

```
##    id table.x          a table.y          b
## 1:  1  table1 -0.5345377  table2  1.0071995
## 2:  2  table1 -0.6252274  table2  0.7192918
## 3:  4    <NA>         NA  table2 -0.6047117
```

4.12 Summary

By now, you should be able to answer the following questions:

- Why do we say that data.table is an enhanced data.frame?
- How to subset by rows or columns? Remember: DT[i, j, by].
- How to add columns?
- How to make operations with different columns?
- Which are the different types of merging?

Even if you were able to answer them, practice:

- Check all vignettes and reference manual from https://cran.r-project.org/web/packages/data.table/
- A really concise cheat sheet:
- https://s3.amazonaws.com/../assets.datacamp.com/img/blog/data+table+cheat+sheet.pdf

4.13 Data.table resources

https://cran.r-project.org/web/packages/data.table/

https://s3.amazonaws.com/../assets.datacamp.com/img/blog/data+table+cheat+sheet.pdf

http://r4ds.had.co.nz/relational-data.html

http://adv-r.had.co.nz/Environments.html

Chapter 5

Tidy data

This chapter is partially adopted from “Introduction to Data Science” by Rafael A. Irizarry (https://rafalab.github.io/dsbook/).

We say that a data table is in *tidy* format if each row represents one observation and columns represent the different variables available for each of these observations. The `murders` dataset is an example of a tidy dataset.

```
##        state abb region population total
## 1    Alabama  AL  South    4779736   135
## 2     Alaska  AK   West     710231    19
## 3    Arizona  AZ   West    6392017   232
## 4   Arkansas  AR  South    2915918    93
## 5 California  CA   West   37253956  1257
## 6   Colorado  CO   West    5029196    65
```

Each row represent a state with each of the five columns providing a different variable related to these states: name, abbreviation, region, population, and total murders.

To see how the same information can be provided in different formats, consider the following example:

This tidy dataset provides fertility rates for two countries across the years. This is a tidy dataset because each row presents one observation with the three variables being country, year, and fertility rate. However, this dataset originally came in another format and was reshaped for the **dslabs** package. Originally, the data was in the following format:

The same information is provided, but there are two important differences in the format: 1) each row includes several observations and 2) one of the variables, year, is stored in the header.

Although not immediately obvious, as you go through the script you will start to appreciate the advantages of working in a framework in which functions use tidy formats for both inputs and outputs. You will see how this permits the data analyst to focus on more important aspects of the analysis rather than the format of the data.

5.1 Tidy data definition

One can briefly summarize the tidy definition in the three following statements:

1. Each **variable** must have its own **column**.
2. Each **observation** must have its own **row**.
3. Each **value** must have its own **cell**.

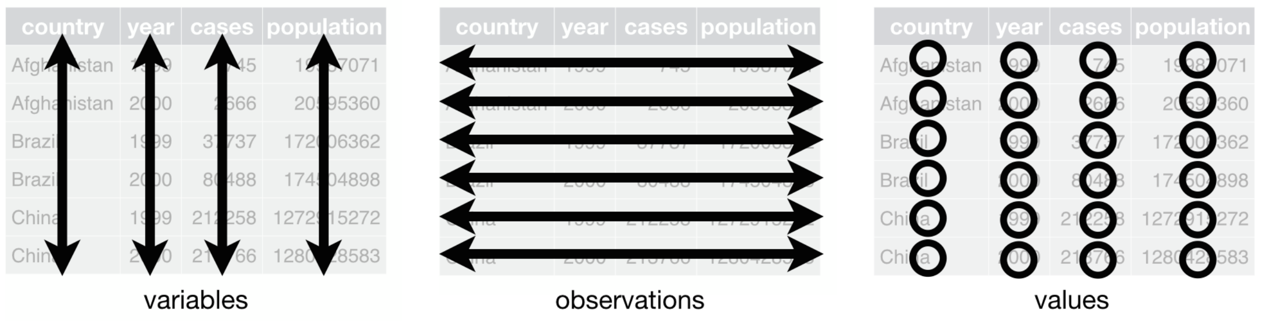


Figure 5.1: **Tidy data table layout.** Each variable has a column, each observation a row and each value a cell.

5.2 Common signs of untidy datasets

On the other hand one can often quickly identify untidy datasets by one or more of the following statements:

- Column headers are values, not variable names.
- Multiple variables are stored in one column.
- Variables are stored in both rows and columns.
- Multiple types of observational units are stored in the same table.
- A single observational unit is stored in multiple tables.

5.3 Advantages of tidy data

- Easier manipulation using data.table commands
- sub-setting by rows
- sub-setting by columns
- `by` operations
- Many other tools work better with tidy data - consistent way of storing data
- example: ggplot2
- Vectorized operations become easier to use

Tidy data can be easily manipulated

```r
head(dt)
```

```
##        country year  cases population
## 1: Afghanistan 1999    745   19987071
## 2: Afghanistan 2000   2666   20595360
## 3:      Brazil 1999  37737  172006362
## 4:      Brazil 2000  80488  174504898
## 5:       China 1999 212258 1272915272
## 6:       China 2000 213766 1280428583
```

For example in the table above we can easily compute the rate of cases within the population or the number of cases per year using the following commands:

```r
# Compute rate per 10,000
dt[, `:=`(rate, cases/population * 10000)]  # vectorized operations; dt is modified
head(dt)
```

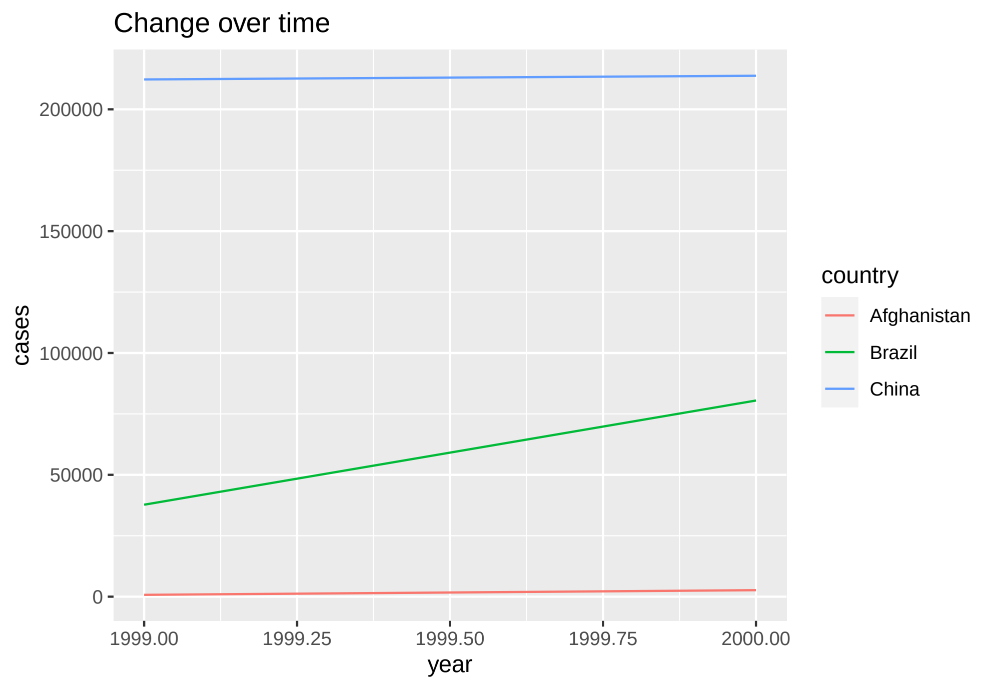
```
##        country year  cases population     rate
## 1: Afghanistan 1999    745   19987071 0.372741
## 2: Afghanistan 2000   2666   20595360 1.294466
## 3:      Brazil 1999  37737  172006362 2.193930
## 4:      Brazil 2000  80488  174504898 4.612363
## 5:       China 1999 212258 1272915272 1.667495
## 6:       China 2000 213766 1280428583 1.669488
```

```
# Compute cases per year
dt[, .(cases = sum(cases)), by = year]  # note that this does not modify dt
```

```
##    year  cases
## 1: 1999 250740
## 2: 2000 296920
```

Additionally, tidy data works better with many packages like ggplot2 which we are going to use in this course.

```
ggplot(dt, aes(year, cases, color = country)) + ggtitle("Change over time") + geom_line()
```



In the remaining part of this chapter we will learn how to transform untidy datasets into tidy ones using data.table functions.

5.3.1 Column headers are values, not variable names

Untidy: 1999 and 2000 are values of the variable *year*.

```
table4a
```

```
##        country   1999   2000
## 1: Afghanistan    745   2666
## 2:      Brazil  37737  80488
## 3:       China 212258 213766
```

This can be solved by using the **data.table** function `melt()`. See the code below. When melting all values in all columns specified in the `measure.vars` argument are gathered into one column whose name can be specified using the `value.name` argument. Additionally a new column named using the argument `variable.name` is created containing all the values which where previously stored in the column names.

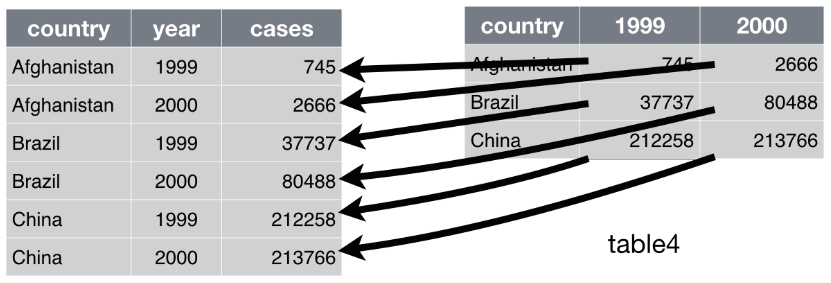


Figure 5.2: Melting the country dataset

```r
melt(table4a, id.vars = "country", measure.vars = c("1999", "2000"), variable.name = "year",
    value.name = "cases")
```

```
##        country year  cases
## 1: Afghanistan 1999    745
## 2:      Brazil 1999  37737
## 3:       China 1999 212258
## 4: Afghanistan 2000   2666
## 5:      Brazil 2000  80488
## 6:       China 2000 213766
```

```r
# would work also without specifying *either* measure.vars OR id.vars
```

5.3.2 Multiple variables are stored in one column

Untidy: multiple variables are stored in the *count* column.

```r
table2
```

```
##         country year       type      count
##  1: Afghanistan 1999      cases        745
##  2: Afghanistan 1999 population   19987071
##  3: Afghanistan 2000      cases       2666
##  4: Afghanistan 2000 population   20595360
##  5:      Brazil 1999      cases      37737
##  6:      Brazil 1999 population  172006362
##  7:      Brazil 2000      cases      80488
##  8:      Brazil 2000 population  174504898
##  9:       China 1999      cases     212258
## 10:       China 1999 population 1272915272
## 11:       China 2000      cases     213766
## 12:       China 2000 population 1280428583
```

This problem can be solved using the dcast function. Here is the help of that function:

```r
## Help
dcast(data, formula, fun.aggregate = NULL, sep = "_", ..., margins = NULL, subset = NULL,
    fill = NULL, drop = TRUE, value.var = guess(data), verbose = getOption("datatable.verbose"))
```

To use it for our case we only need to specify which column is the key in the “formula” and which value should be spread.

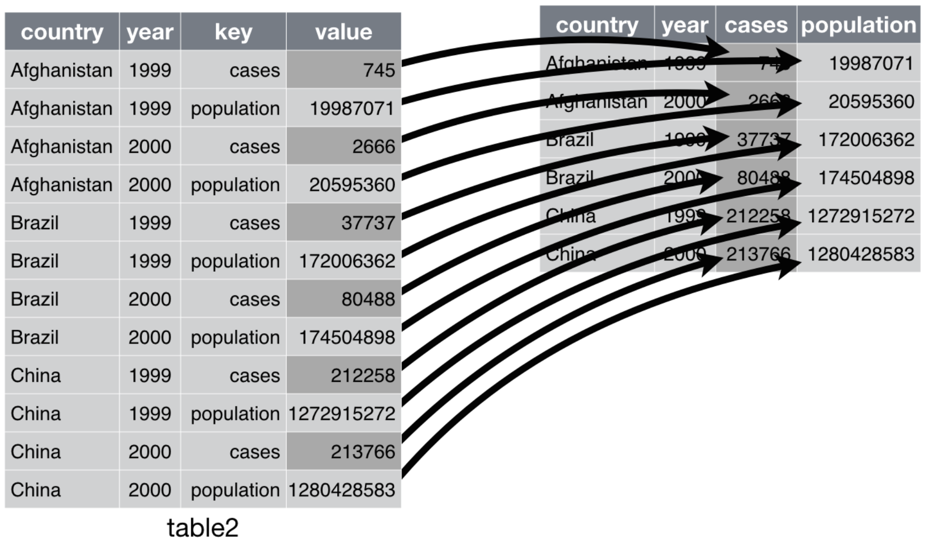

country
year
key
value
Afghanistan 1999 cases 745
Afghanistan 1999 population 19987071
Afghanistan 2000 cases 2666
Afghanistan 2000 population 20595360
Brazil 1999 cases 37737
Brazil 1999 population 172006362
Brazil 2000 cases 80488
Brazil 2000 population 174504898
China 1999 cases 212258
China 1999 population 1272915272
China 2000 cases 213766
China 2000 population 1280428583
table2
country
year
cases
population
Afghanistan 1999 745 19987071
Afghanistan 2000 2666 20595360
Brazil 1999 37737 172006362
Brazil 2000 80488 174504898
China 1999 212258 1272915272
China 2000 213766 1280428583

Figure 5.3: Decasting the country dataset

```
dcast(table2, ... ~ type, value.var = "count")
```

```
##        country year  cases population
## 1: Afghanistan 1999    745   19987071
## 2: Afghanistan 2000   2666   20595360
## 3:      Brazil 1999  37737  172006362
## 4:      Brazil 2000  80488  174504898
## 5:       China 1999 212258 1272915272
## 6:       China 2000 213766 1280428583
```

5.4 Separating and Uniting (1 <-> more variables)

Typical problem:

1. One column contains multiple variables
2. Multiple columns contain one variable

Untidy datasets

```
## One column contains multiple variables
print(table3)
```

```
##        country year              rate
## 1: Afghanistan 1999      745/19987071
## 2: Afghanistan 2000     2666/20595360
## 3:      Brazil 1999   37737/172006362
## 4:      Brazil 2000   80488/174504898
## 5:       China 1999 212258/1272915272
## 6:       China 2000 213766/1280428583
```

```
## Multiple columns contain one variable
print(table5)
```

```
##        country century year              rate
## 1: Afghanistan      19   99      745/19987071
## 2: Afghanistan      20   00     2666/20595360
## 3:      Brazil      19   99   37737/172006362
## 4:      Brazil      20   00   80488/174504898
## 5:       China      19   99 212258/1272915272
## 6:       China      20   00 213766/1280428583
```

Solution in R:

1) variable -> multiple variables

- `tidyr::separate()`

2) multiple variables -> 1 variables

- `tidyr::unite()`

other useful functions:

- `data.table::tstrsplit`, `strsplit`, `paste`, `substr`

To separate 1 variable to multiple variables we use `tidyr::separate()`.

```
separate(data, col, into, sep = "[^[:alnum:]]+", remove = TRUE, convert = FALSE,
    extra = "warn", fill = "warn", ...)
```

```
table3
```

```
##         country year              rate
## 1: Afghanistan 1999      745/19987071
## 2: Afghanistan 2000     2666/20595360
## 3:      Brazil 1999   37737/172006362
## 4:      Brazil 2000   80488/174504898
## 5:       China 1999 212258/1272915272
## 6:       China 2000 213766/1280428583
```

```
separate(table3, col = rate, into = c("cases", "population"))
```

```
##         country year  cases population
## 1: Afghanistan 1999    745   19987071
## 2: Afghanistan 2000   2666   20595360
## 3:      Brazil 1999  37737  172006362
## 4:      Brazil 2000  80488  174504898
## 5:       China 1999 212258 1272915272
## 6:       China 2000 213766 1280428583
```

```
separate(table3, col = rate, into = c("cases", "population")) %>% class
```

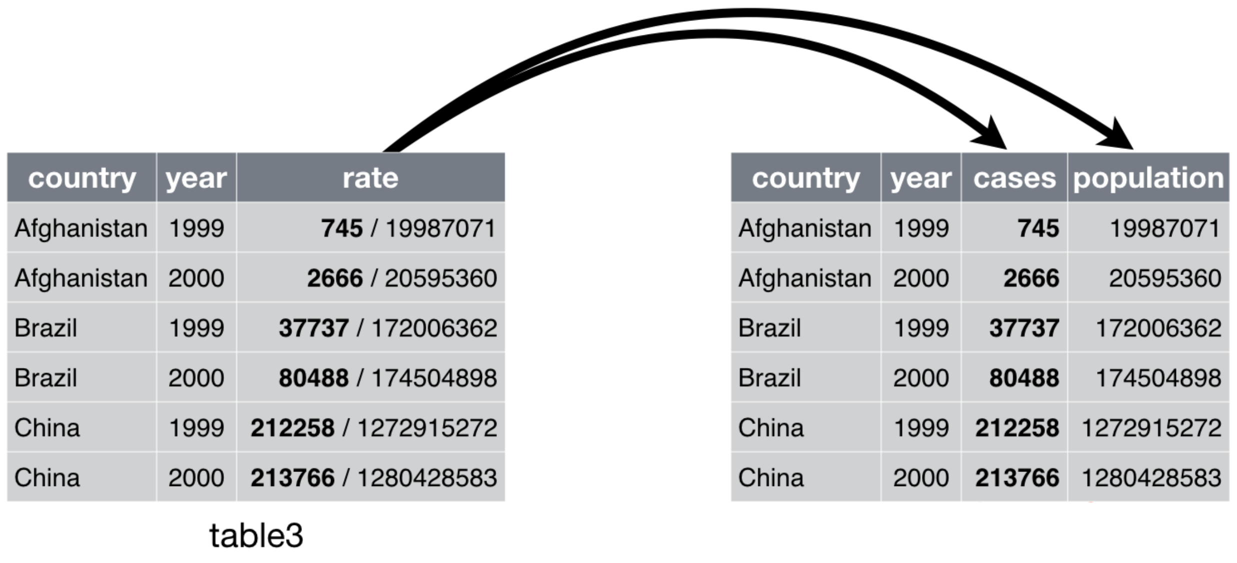
```
## [1] "data.table" "data.frame"
```

To unite multiple variables to 1 variable we use `tidyr::unite()`.

```
unite(data, col, ..., sep = "_", remove = TRUE)
```

```
table5
```

```
##         country century year              rate
## 1: Afghanistan      19   99      745/19987071
## 2: Afghanistan      20   00     2666/20595360
## 3:      Brazil      19   99   37737/172006362
## 4:      Brazil      20   00   80488/174504898
## 5:       China      19   99 212258/1272915272
## 6:       China      20   00 213766/1280428583
```



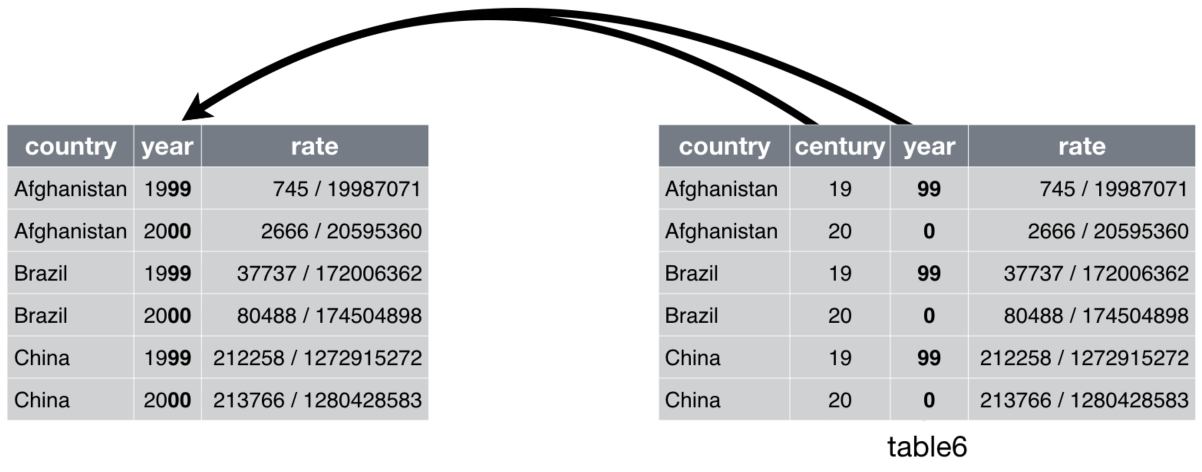
country	year	rate
Afghanistan	1999	**745** / 19987071
Afghanistan	2000	**2666** / 20595360
Brazil	1999	**37737** / 172006362
Brazil	2000	**80488** / 174504898
China	1999	**212258** / 1272915272
China	2000	**213766** / 1280428583

country	year	cases	population
Afghanistan	1999	**745**	19987071
Afghanistan	2000	**2666**	20595360
Brazil	1999	**37737**	172006362
Brazil	2000	**80488**	174504898
China	1999	**212258**	1272915272
China	2000	**213766**	1280428583

Figure 5.4: Seperated country dataset

```r
unite(table5, col = new, century, year, sep = "")
```

```
##        country  new            rate
## 1: Afghanistan 1999     745/19987071
## 2: Afghanistan 2000    2666/20595360
## 3:      Brazil 1999   37737/172006362
## 4:      Brazil 2000   80488/174504898
## 5:       China 1999 212258/1272915272
## 6:       China 2000 213766/1280428583
```



country	year	rate
Afghanistan	1**999**	745 / 19987071
Afghanistan	2**000**	2666 / 20595360
Brazil	1**999**	37737 / 172006362
Brazil	2**000**	80488 / 174504898
China	1**999**	212258 / 1272915272
China	2**000**	213766 / 1280428583

country	century	year	rate
Afghanistan	19	**99**	745 / 19987071
Afghanistan	20	**0**	2666 / 20595360
Brazil	19	**99**	37737 / 172006362
Brazil	20	**0**	80488 / 174504898
China	19	**99**	212258 / 1272915272
China	20	**0**	213766 / 1280428583

Figure 5.5: United country dataset

5.4.0.0.1 List of data.tables

```r
split(table1, table1$country) %>% lapply(. %>% subset(select = -country) %>% as.data.table)
```

```
## $Afghanistan
##    year cases population     rate
## 1: 1999   745   19987071 0.372741
## 2: 2000  2666   20595360 1.294466
##
## $Brazil
##    year cases population     rate
## 1: 1999 37737  172006362 2.193930
## 2: 2000 80488  174504898 4.612363
##
## $China
##    year  cases population     rate
## 1: 1999 212258 1272915272 1.667495
## 2: 2000 213766 1280428583 1.669488
```

To sum up:

- In a tidy dataset, each variable must have its own column
- Each row corresponds to one unique observation
- Each cell contains a single value
- Tidy datasets are easier to work with
- Data.table library has functions to transform untidy datasets to tidy

Words to live by

Happy families are all alike; every unhappy family is unhappy in its own way.

- Leo Tolstoy

Tidy datasets are all alike, but every messy dataset is messy in its own way.

- Hadley Wickham

5.5 Non-tidy data

- Performance advantage using certain functions
  - `colSums()` or `heatmap()` on matrices
- Field convention
- Memory efficiency
  - don't worry, you should be fine with tidy-data in `data.table`

Interesting blog post:

- http://simplystatistics.org/2016/02/17/non-tidy-data/

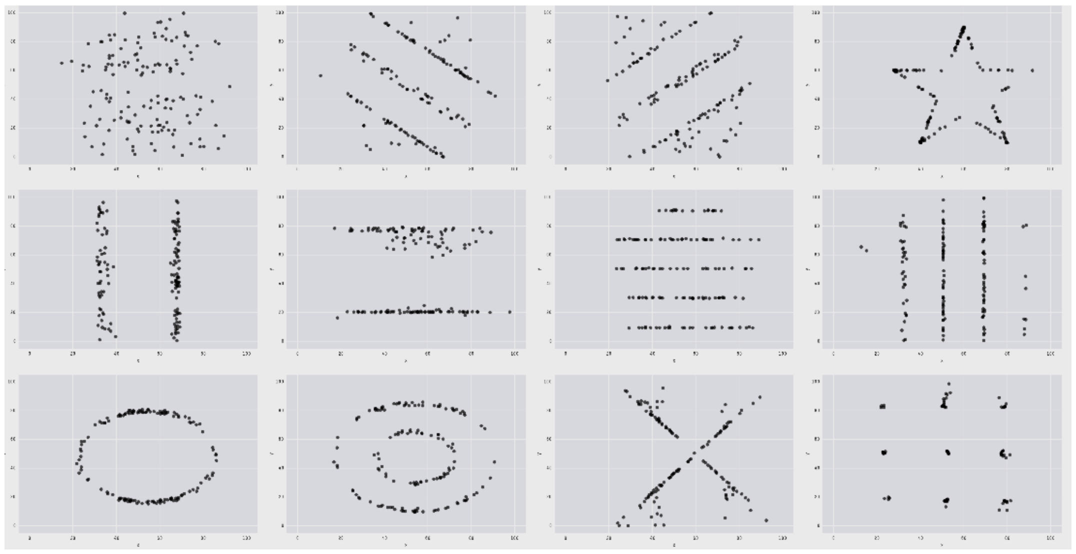
Chapter 6

Low dimensional visualizations

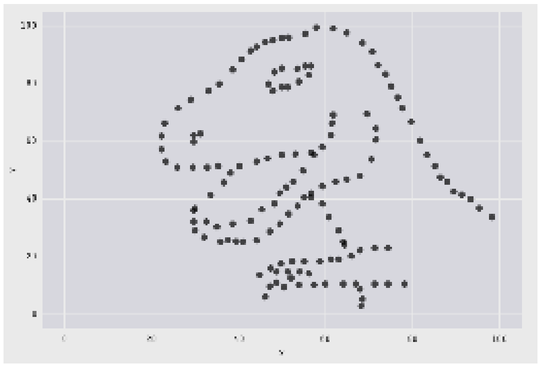
6.1 Why plotting?

With the increasing amounts of data, it's become increasingly difficult to manage and make sense of it all. It becomes nearly impossible for a single person to go through data line-by-line and see distinct patterns and make observations. In this context, data visualization becomes crucial for gaining insight into data that traditional descriptive statistics cannot. For instance, plotting can give hints about bugs in our code (or even in the data!) and can help us to develop and improve methods and models.

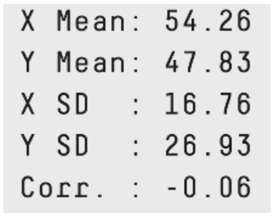
Do you see a pattern in these plots?



Maybe now?



All those plots, including the infamous datasaurus, actually share the same statistics!



```
X Mean: 54.26
Y Mean: 47.83
X SD  : 16.76
Y SD  : 26.93
Corr. : -0.06
```

When only looking at the statistics, we would have probably wrongly assumed that the datasets were identical. This illustration highlights why it is important to visualize data and not just rely on descriptive statistics.

We can consider another example given by a vector `height` containing (hypothetical) height measurements for 500 adults in Germany:

```
length(height)
```

```
## [1] 500
```

```
head(height, n = 20)
```

```
##  [1] 1.786833 1.715319 1.789841 1.787259 1.748659 1.660702 1.688214 1.716738
##  [9] 1.729740 1.838209 1.764362 1.694045 1.678355 1.716974 1.716535 1.711482
## [17] 1.741350 1.689864 1.749606 1.674311
```

Calculating the mean height returns the following output:

```
mean(height)
```

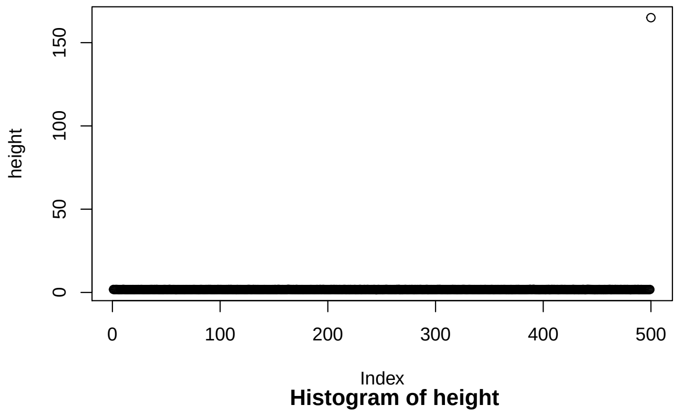
```
## [1] 2.056583
```

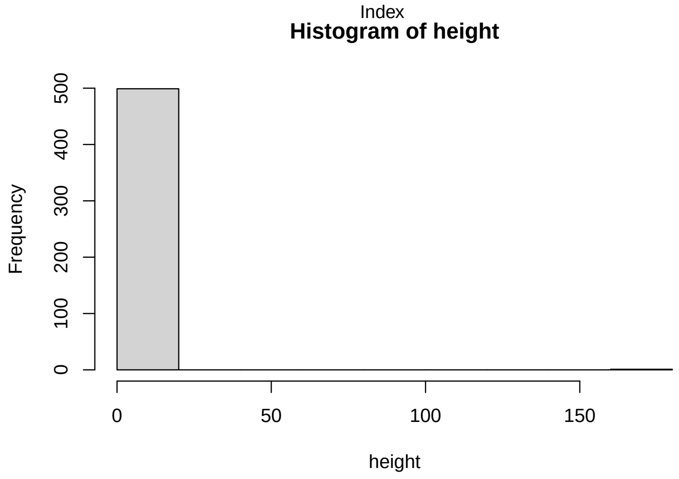
Wait... what?

What happened?

Assuming that adults in Germany are actually not exceptionally tall, we can plot the data for investigating this situation.

```
plot(height)
hist(height)
```





Interestingly, there is an outlier in our data! One particular person seems to have a height above 150 meters that is obviously wrong. As a result, it inflates our mean giving us the false impression that Germans are exceptionally tall.

A quick way to fix our dataset is to remove our outlier.

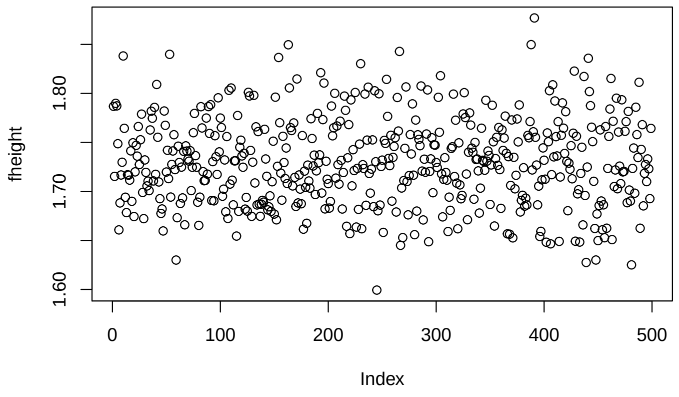
```
fheight <- height[height < 3]
```

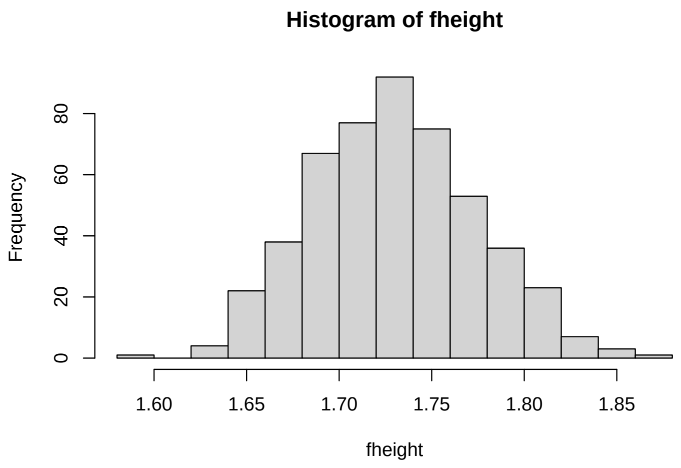
Now our plotted data seems more realistic and the mean height makes sense.

```
mean(fheight)
```

```
## [1] 1.730043
```

```
plot(fheight)
hist(fheight)
```





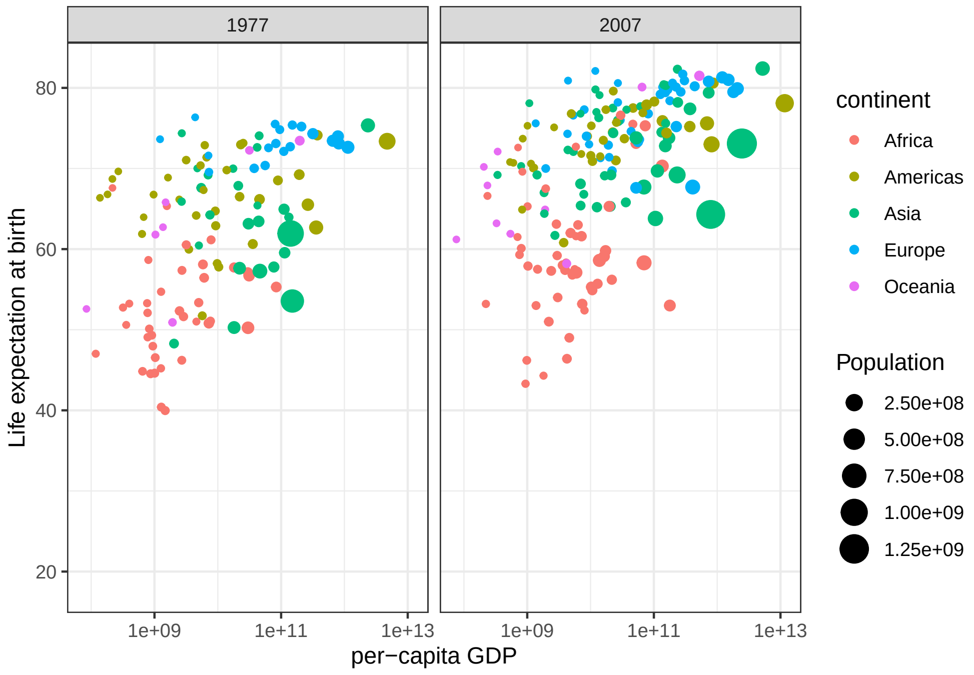
6.2 Grammar of graphics

Plotting with the help of the package `ggplot2` has become widely used by R programmers. The plotting logic of `ggplot2` is known as the grammar of graphics. The grammar of graphics is a visualization theory developed by Leland Wilkinson in 1999. It has influenced the development of graphics and visualization libraries alike and is based on 3 key principles:

- Separation of data from aesthetics (e.g. x and y-axis, color-coding)
- Definition of common plot/chart elements (e.g. scatter plots, box-plots, etc.)
- Composition of these common elements (one can combine elements as layers)

Now let's try to create a sophisticated example based on the R package `ggplot2` for visualizing the relationship of between the per-capita GDP `gdp` and the life expectancy at birth `life_expectancy` for every country from the dataset `gapminder`. We want to compare this relationship for the years 1977 and 2007:

```
# install.packages('gapminder')
library(gapminder)
gm_dt <- as.data.table(gapminder)[year %in% c(1977, 2007)]

ggplot(data = gm_dt, aes(x = gdp, y = life_expectancy)) + geom_point(aes(color = continent,
    size = population)) + facet_grid(~year) + scale_x_log10() + labs(y = "Life expectation at birth",
    x = "per-capita GDP", size = "Population") + mytheme
```



We may, for instance, use such visualization to find differences in the life expectancy of each country and each continent.

Now... how do we create such a sophisticated plot step by step? First, we will introduce the major components of grammar of graphics that will allow us to create and understand the previous example.

6.2.1 Major components of the grammar of graphics

The following components are considered in the context of the grammar of graphics:

Data: `data.table` (or `data.frame`) object where columns correspond to variables

Aesthetics: visual characteristics that represent data (`aes`) - e.g. position, size, color, shape, transparency, fill

Layers: geometric objects that represent data (`geom_`) - e.g. points, lines, polygons, ...

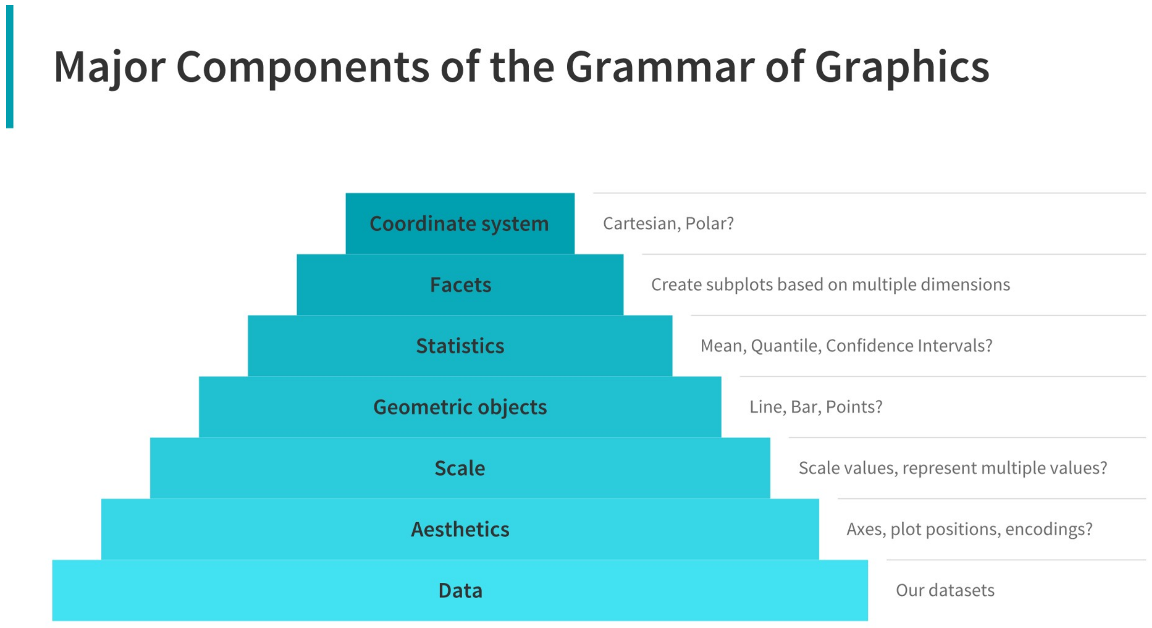
Scales: for each aesthetic, describes how visual characteristic is converted to display values (`scale_`) - e.g. log scales, color scales, size scales, shape scales, ...

Facets: describes how data is split into subsets and displayed as multiple subgraphs (`facet_`)

Stats: statistical transformations that typically summarize data (`stat`) - e.g. counts, means, medians, regression lines, ...

Coordinate system: describes 2D space that data is projected onto (`coord_`) - e.g. Cartesian coordinates, polar coordinates, map projections, ...

The following illustration represents the abstraction of grammar of graphics:


Major Components of the Grammar of Graphics
Coordinate system
Cartesian, Polar?
Facets
Create subplots based on multiple dimensions
Statistics
Mean, Quantile, Confidence Intervals?
Geometric objects
Line, Bar, Points?
Scale
Scale values, represent multiple values?
Aesthetics
Axes, plot positions, encodings?
Data
Our datasets

6.2.2 Defining the data and layers

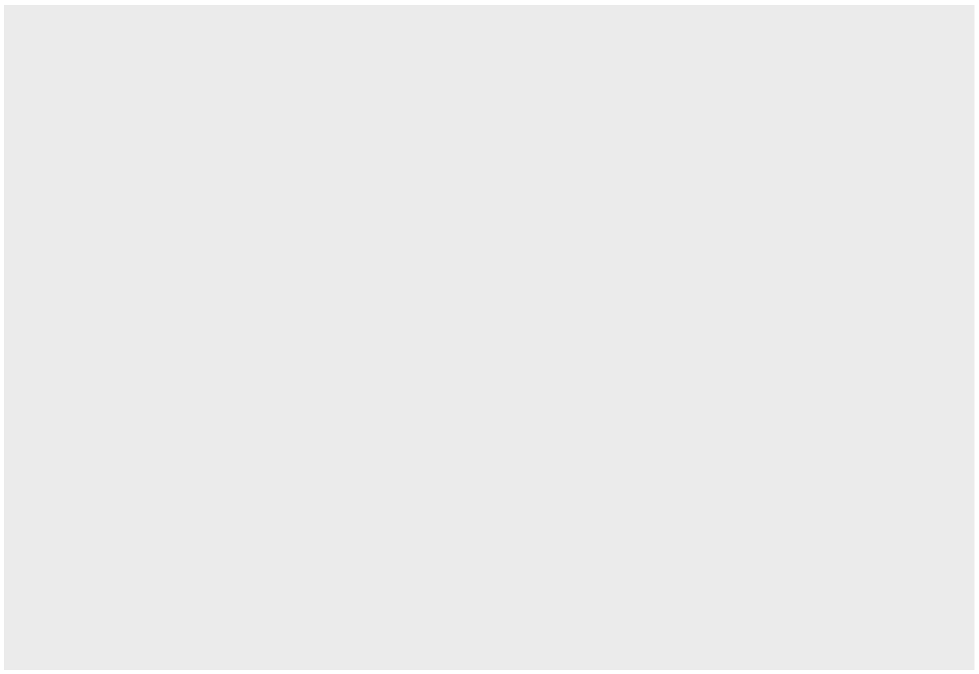
In our example, we consider the `gapminder` dataset, which serves as the data component of our visualization. First, we have a quick look at the data. We want to plot the variable `life_expectancy` against the variable `gdp` so we better have a look at the first lines of the data:

```
head(gm_dt[, .(country, continent, gdp, life_expectancy, year)])
```

```
##                country continent          gdp life_expectancy year
## 1:             Albania    Europe           NA           70.54 1977
## 2:             Algeria    Africa  29829518900           57.13 1977
## 3:              Angola    Africa           NA           45.12 1977
## 4: Antigua and Barbuda  Americas    268265361           69.64 1977
## 5:           Argentina  Americas 193500979043           69.24 1977
## 6:             Armenia      Asia           NA           72.44 1977
```

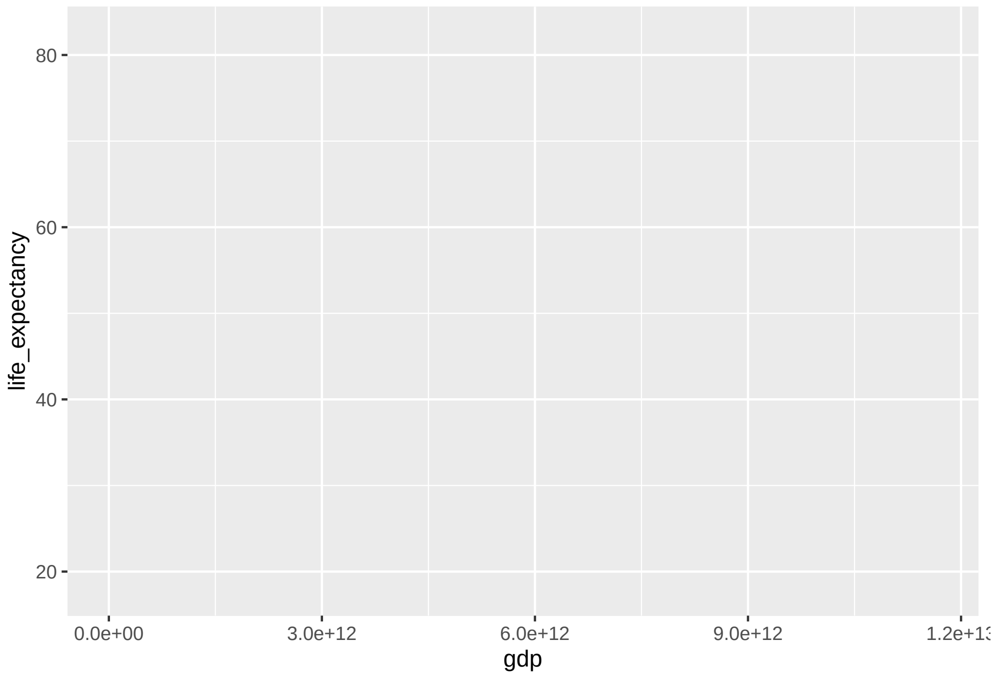
For starting with the visualization we initiate a `ggplot` object which generates a plot with background:

```
ggplot()
```



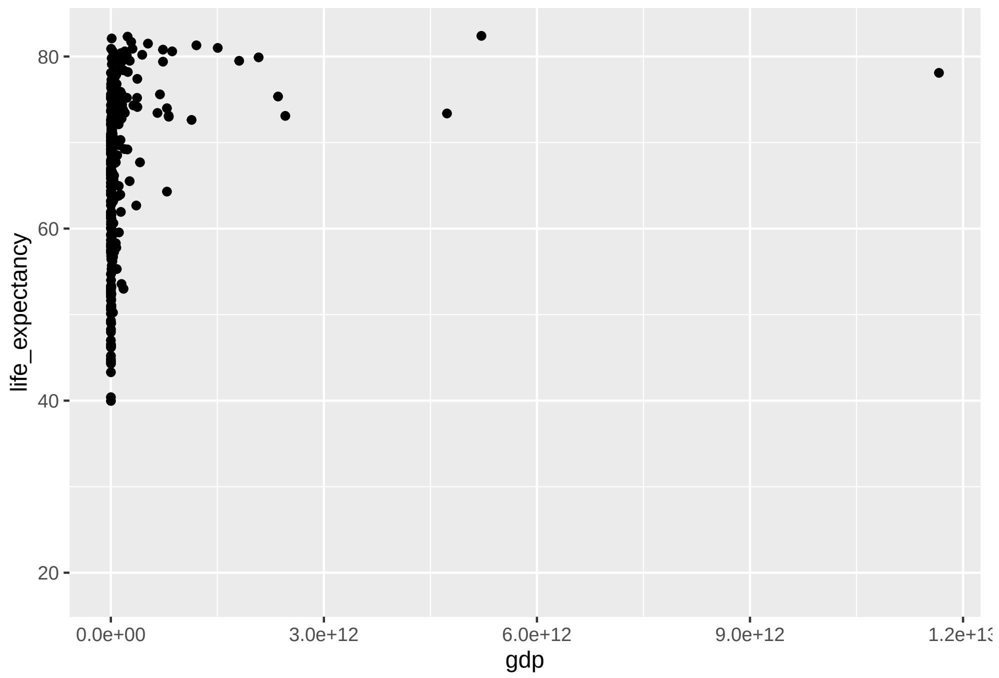
Next, we can define the data to be plotted, which needs to be a `data.table` (or `data.frame`) object and the `aes()` function. This `aes()` function defines which columns in the `data.table` object map to `x` and `y` coordinates and if they should be colored or have different shapes and sizes based on the values in a different column. These elements are called “aesthetic” elements, which we observe in the plot.

```
ggplot(data = gm_dt, aes(x = gdp, y = life_expectancy))
```



As we can see, we obtain a plot with labeled axis and ranges. We want to visualize the data with a simple **scatter plot**. In a scatter plot, the values of two variables are plotted along two axes. Each pair of values is represented as a point. In R, a scatter plot can be plotted with `ggplot2` using the function `geom_point`. We want to construct a scatter plot containing the `gdp` on the x-axis and the `life_expectancy` on the y-axis. For this we combine the function `geom_point()` to the previous line of code with the operator `+`:

```
ggplot(data = gm_dt, aes(x = gdp, y = life_expectancy)) + geom_point()
```



One of the advantages of plotting with `ggplot` is that it returns an object which can be stored (e.g. in a variable called `p`). The stored object can be further edited.

```
p <- ggplot(data = gm_dt, aes(x = gdp, y = life_expectancy)) + geom_point()
```

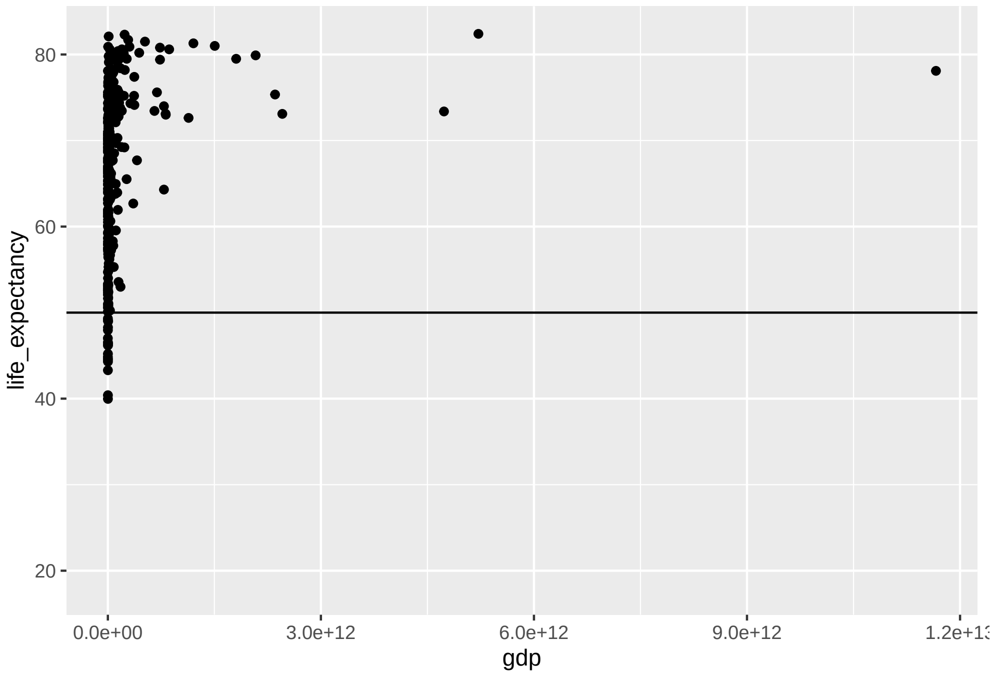
We can inspect the names of elements of the stored object with `names()`:

```
names(p)
```

```
## [1] "data"        "layers"      "scales"      "mapping"     "theme"
## [6] "coordinates" "facet"       "plot_env"    "labels"
```

We can also save the `ggplot` object with the help of the function `saveRDS()`. Then, we can read the saved object again with the help of the function `readRDS()` and add a horizontal line at `y=50` to the plot:

```
saveRDS(p, "../extdata/my_first_plot.rds")
p <- readRDS("../extdata/my_first_plot.rds")
p + geom_hline(yintercept = 50)
```

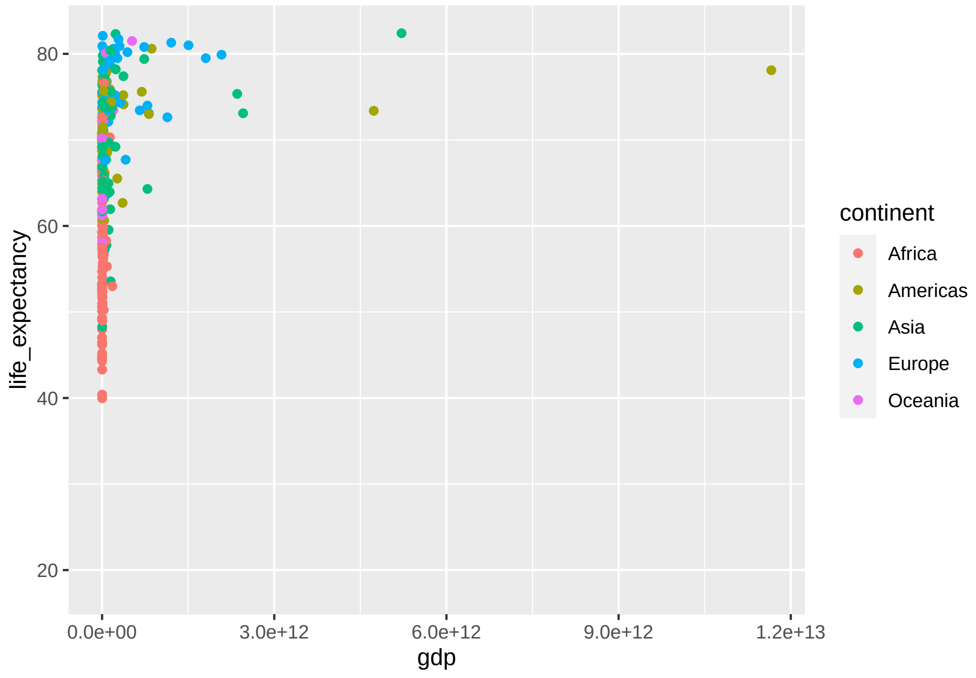


6.2.3 Mapping of aesthetics

6.2.3.1 Mapping of color, shape and size

We can easily map variables to different colors, sizes or shapes depending on the value of the specified variable. To assign each point to its corresponding continent, we can define the variable `continent` as the `color` attribute in `aes()` as follows:

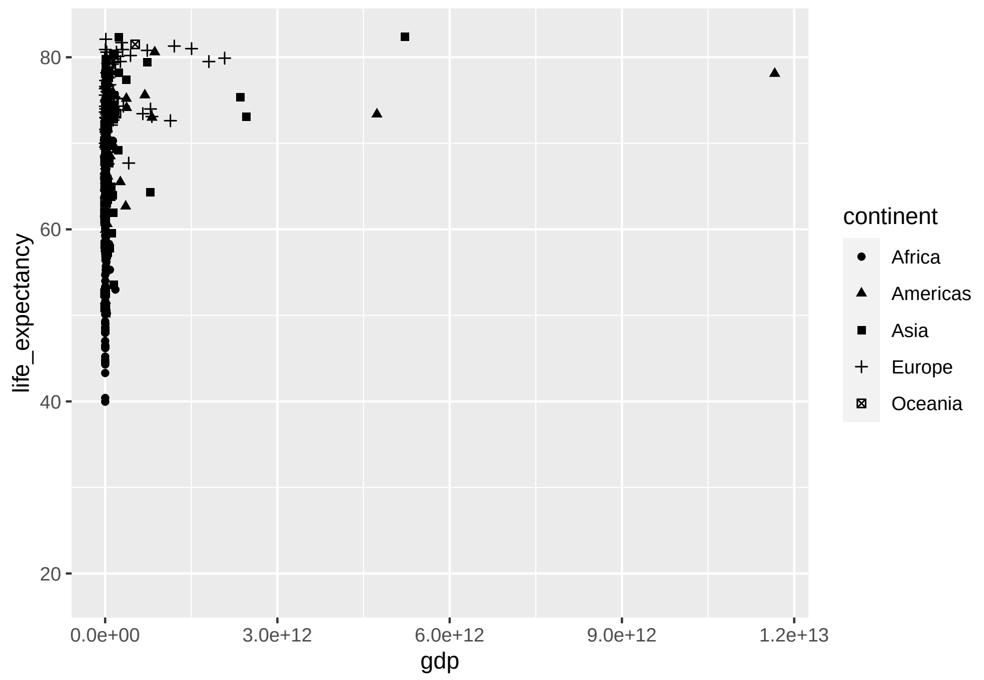
```r
ggplot(data = gm_dt, aes(x = gdp, y = life_expectancy, color = continent)) + geom_point()
```



American `color` or British `colour` are both acceptable as the argument specification.

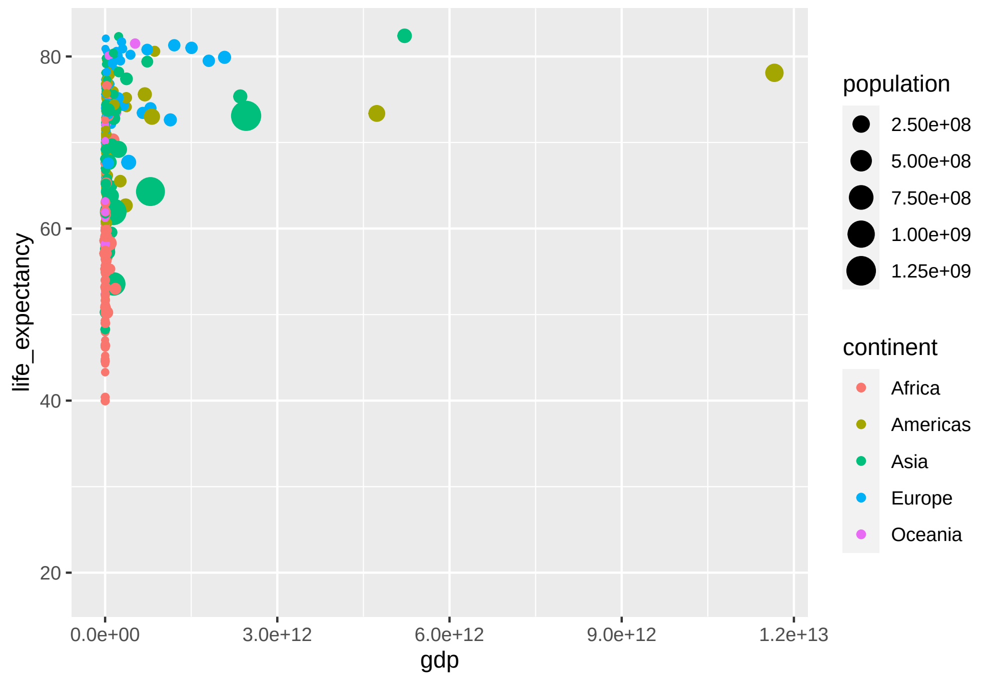
Instead of color, we can also use different shapes for characterizing the different continents in the scatter plot. For this we specify the `shape` argument in `aes()` as follows:

```
ggplot(data = gm_dt, aes(x = gdp, y = life_expectancy, shape = continent)) + geom_point()
```



Additionally, we distinguish the population of each country by giving a size to the points in the scatter plot:

```
ggplot(data = gm_dt, aes(x = gdp, y = life_expectancy, color = continent, size = population)) +
    geom_point()
```

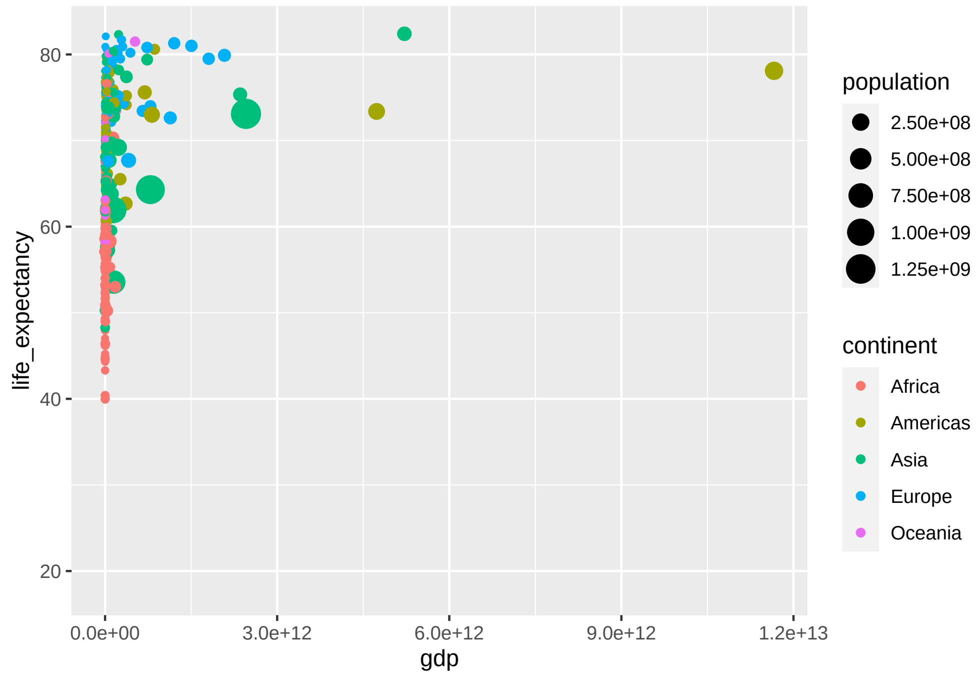


6.2.3.2 Global versus individual mapping

Mapping of aesthetics in `aes()` can be done globally or at individual layers.

For instance, in the previous plot we define the variables `gdp` and `life_expectancy` in the `aes()` function inside `ggplot()` for a **global** definition. Global mapping is inherited by default to all geom layers (`geom_point` in the previous example), while mapping at **individual** layers is only recognized at that layer. For example, we define the `aes(x=gdp, y=life_expectancy)` globally, but the color attributes only locally for the layer `geom_point`:

```
ggplot(data = gm_dt, aes(x = gdp, y = life_expectancy)) + geom_point(aes(color = continent,
    size = population))
```

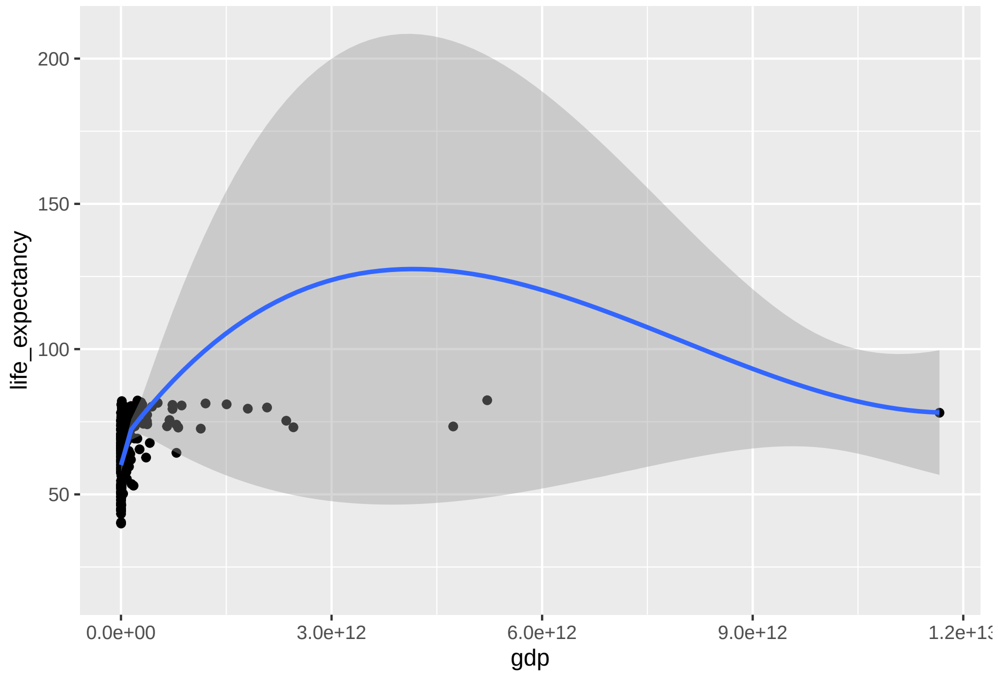


Note that individual layer mapping cannot be recognized by other layers. For instance, we can add another layer for smoothing with `stat_smooth()`.

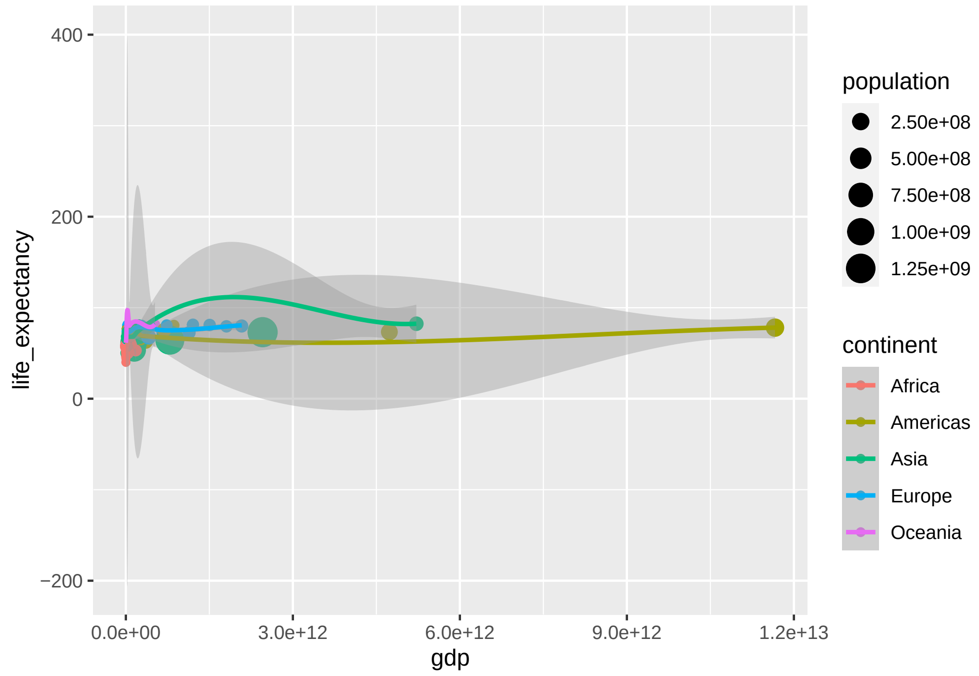
```
# this doesn't work as stat_smooth didn't know aes(x , y)
ggplot(data = gm_dt) + geom_point(aes(x = gdp, y = life_expectancy)) + stat_smooth()
```

```
## Error: stat_smooth requires the following missing aesthetics: x and y
```

```
# this would work but too redundant
ggplot(data = gm_dt) + geom_point(aes(x = gdp, y = life_expectancy)) + stat_smooth(aes(x = gdp,
    y = life_expectancy))
```



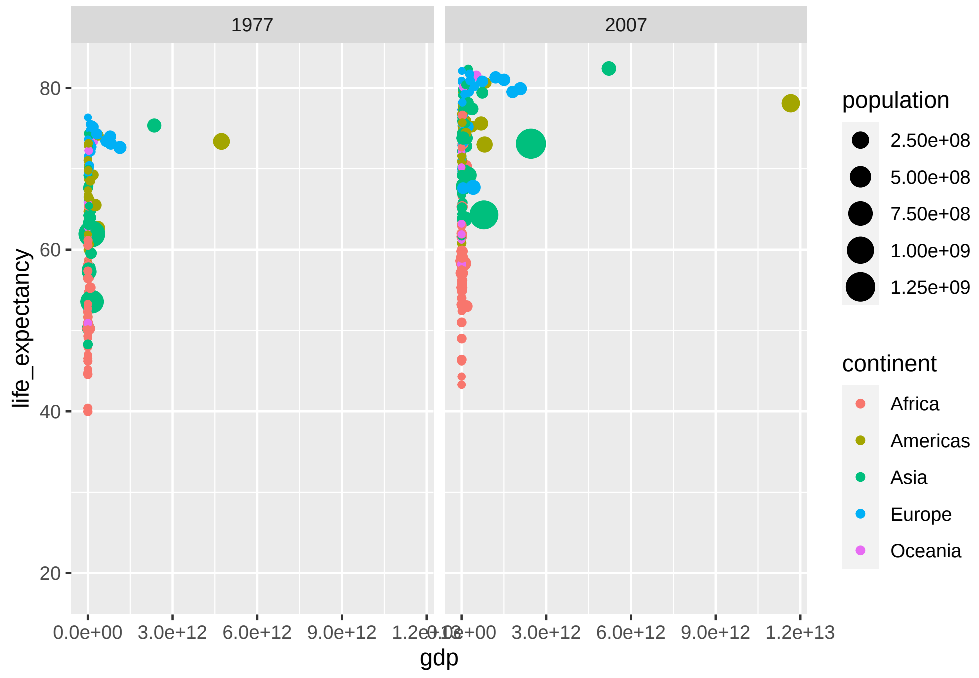
```
# the common aes(x, y) shared by all the layers can be put in the ggplot()
ggplot(data = gm_dt, aes(x = gdp, y = life_expectancy, color = continent)) + geom_point(aes(size = populati
    stat_smooth()
```



6.2.4 Facets, axes and labels

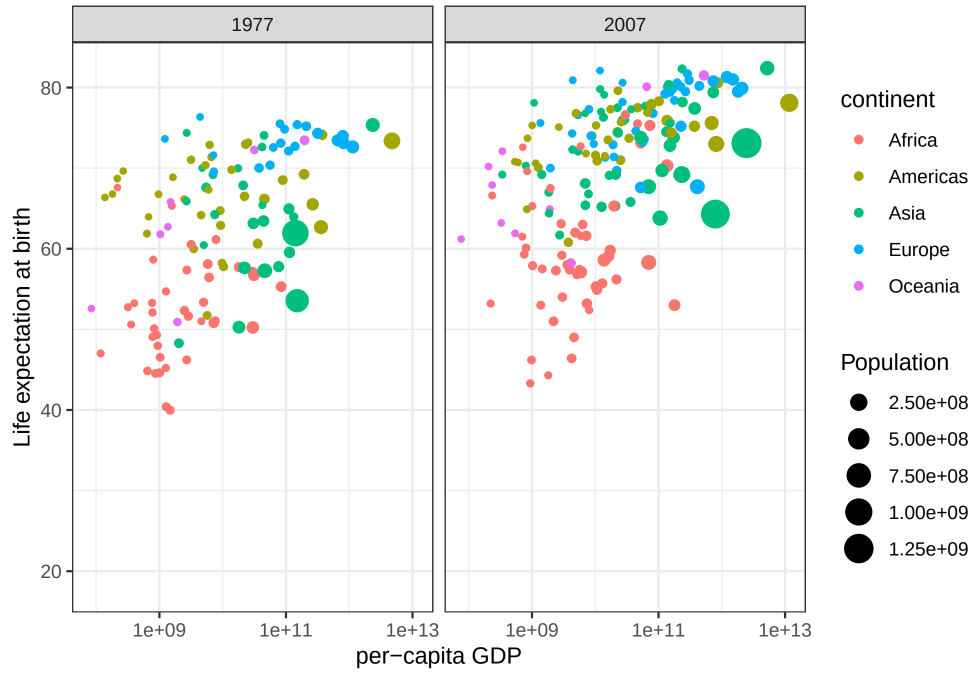
For comparing the data from the year 1977 with the data from 2007, we can add a facet with `facet_wrap()`:

```
ggplot(data = gm_dt, aes(x = gdp, y = life_expectancy, color = continent, size = population)) +
    geom_point() + facet_wrap(~year)
```



For a better visualization of the data points, we can consider log scaling, which we will describe more in detail later. Finally, we can adapt the axes labels of the plot with `labs()` and define a theme of our plot:

```
mysize <- 15
mytheme <- theme(axis.title = element_text(size = mysize), axis.text = element_text(size = mysize),
    legend.title = element_text(size = mysize), legend.text = element_text(size = mysize)) +
    theme_bw()

ggplot(data = gm_dt, aes(x = gdp, y = life_expectancy)) + geom_point(aes(color = continent,
    size = population)) + facet_grid(~year) + scale_x_log10() + labs(x = "per-capita GDP",
    y = "Life expectation at birth", size = "Population") + mytheme
```



We remark here that `ggplot2` allows many further adaptions to plots, such as specifying axis breaks and limits. Some of these are covered in the appendix at the end of this script.

6.3 Different types of one- and two-dimensional plots

In the previous examples, we had a look at scatter plots which are suitable for plotting the relationship between two continuous variables. However, there are many more types of plots (e.g. histograms, boxplots) which can be used for plotting in different scenarios. Mainly, we distinguish between plotting one or two variables and whether the variables are continuous or discrete.

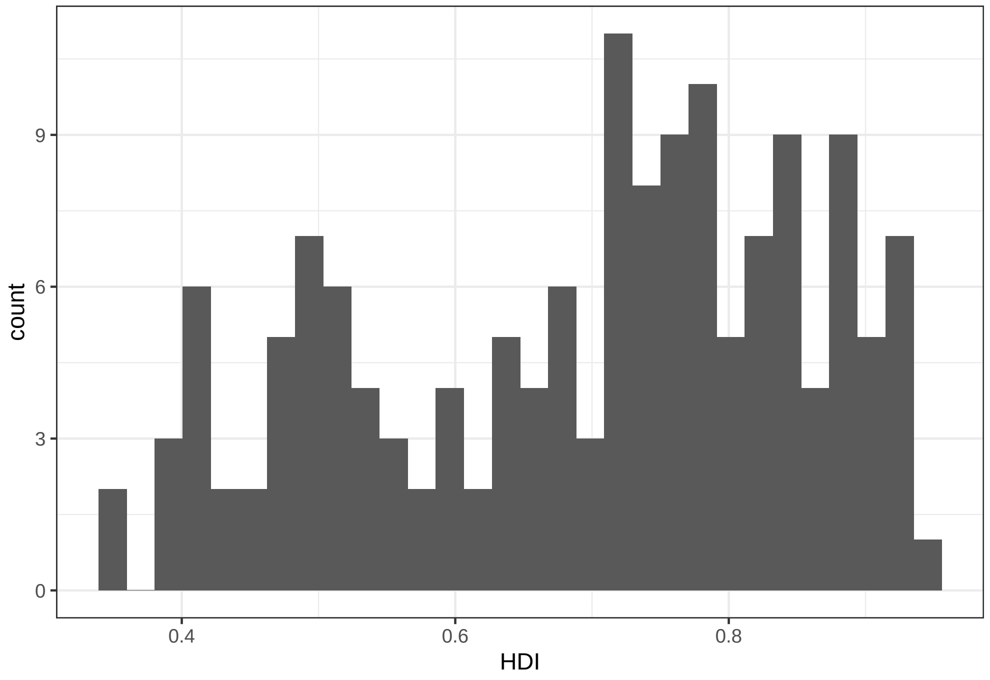
6.3.1 Plots for one single continuous variable

6.3.1.1 Histograms

A histogram represents the frequencies of values of a variable bucketed into ranges. It takes as input numeric variables only. A histogram is similar to a bar plot but the difference is that it groups the values into continuous ranges. Each bar in a histogram represents the height of the number of values present in that range. Each bar of the histogram is called a bin.

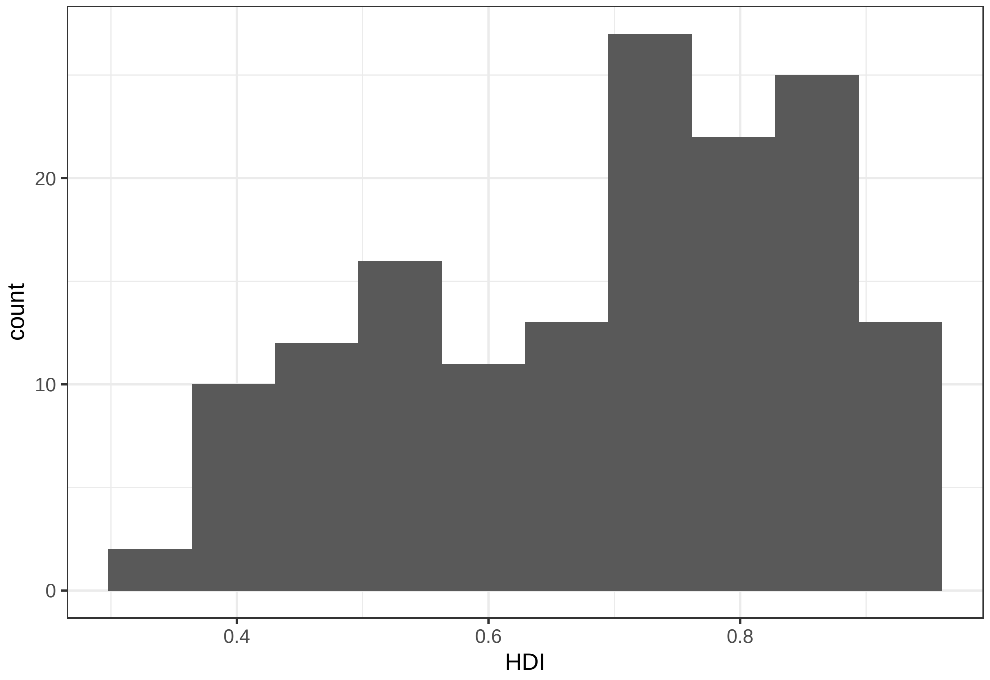
We can construct a histogram of the Human Development Index (HDI) in the `ind` dataset with the function `geom_histogram()`:

```
ggplot(ind, aes(HDI)) + geom_histogram() + mytheme
```



By default, the number of bins in `ggplot2` is 30. We can simply change this by defining the number of desired bins in the `bins` argument of the `geom_histogram()` function:

```
ggplot(ind, aes(HDI)) + geom_histogram(bins = 10) + mytheme
```

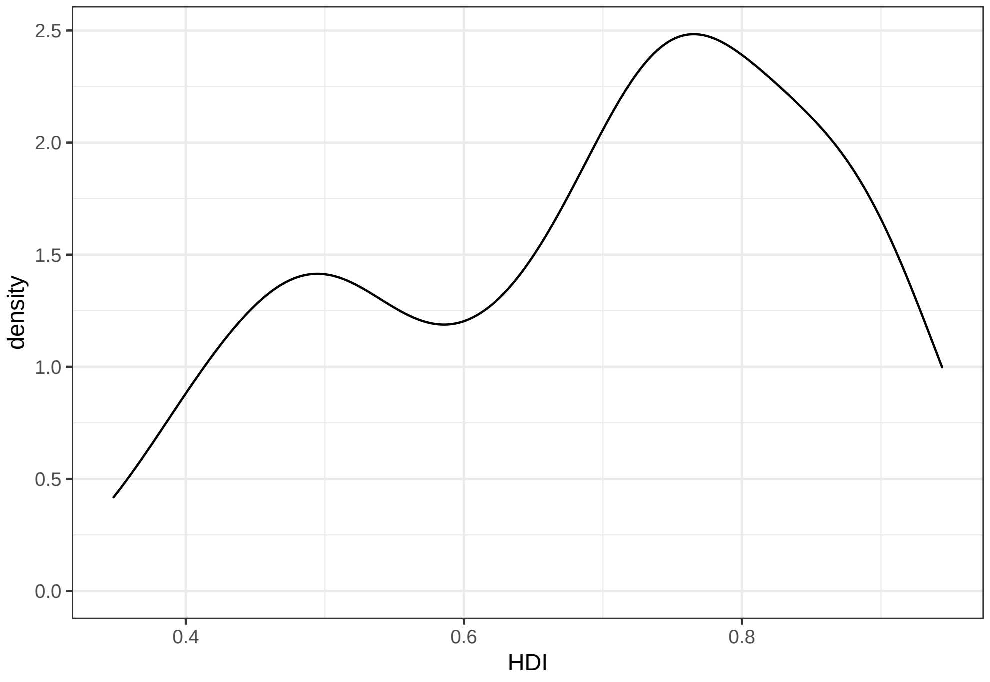


6.3.1.2 Density plots

Histograms are sometimes not optimal to investigate the distribution of a variable due to discretization effects during the binning process. A variation of histograms is given by density plots. They are used to represent the distribution of a numeric variable. These distribution plots are typically obtained by kernel density estimation to smoothen out the noise. Thus, the plots are smooth across bins and are not affected by the number of bins, which helps create a more defined distribution shape.

As an example, we can visualize the distribution of the Human Development Index (HDI) in the `ind` dataset by means of a density plot with `geom_density()`:

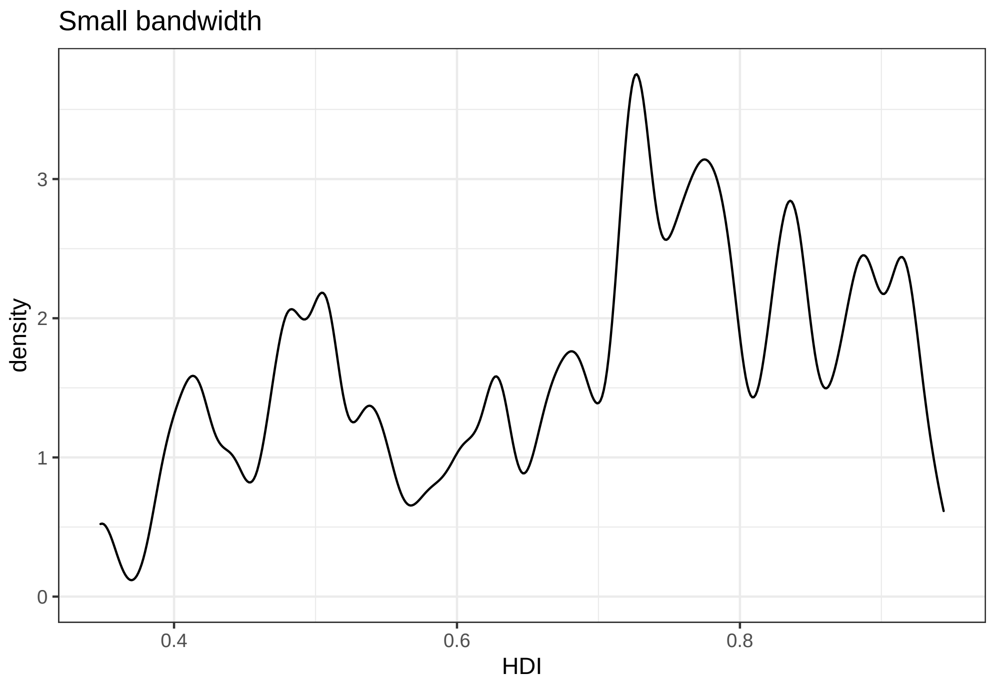
```r
ggplot(ind, aes(HDI)) + geom_density() + mytheme
```



The `bw` argument of the `geom_density()` function allows to tweak the bandwidth of a density plot manually. The default option is a bandwidth rule, which is usually a good choice.

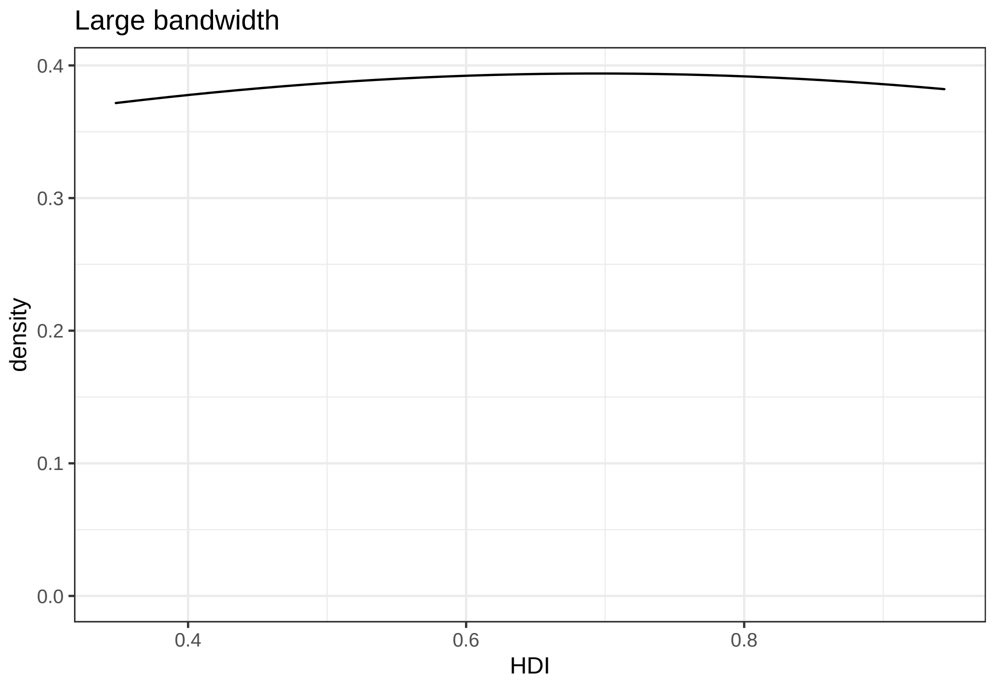
Setting a small bandwidth on the previous plot has a huge impact on the plot:

```
ggplot(ind, aes(HDI)) + geom_density(bw = 0.01) + ggtitle("Small bandwidth") + mytheme
```



Setting a large bandwidth has also a huge impact on the plot:

```
ggplot(ind, aes(HDI)) + geom_density(bw = 1) + ggtitle("Large bandwidth") + mytheme
```



Thus, we should be careful when changing the bandwidth, since we can get a wrong impression from the distribution of a continuous variable.

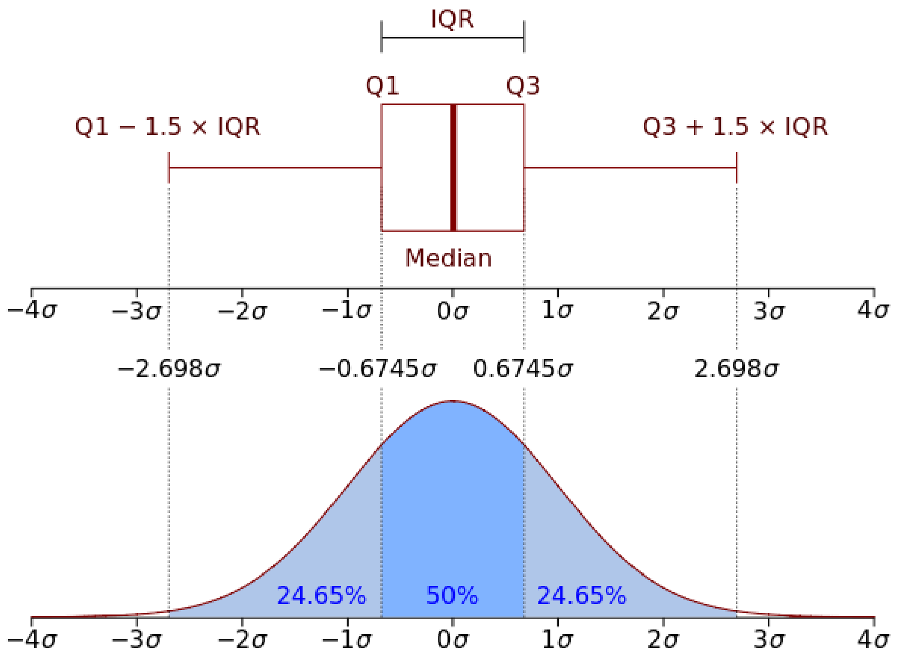
6.3.1.3 Boxplots

Box plots can give a good graphical insight into the distribution of the data. They show the median, quantiles, and how far the extreme values are from most of the data.

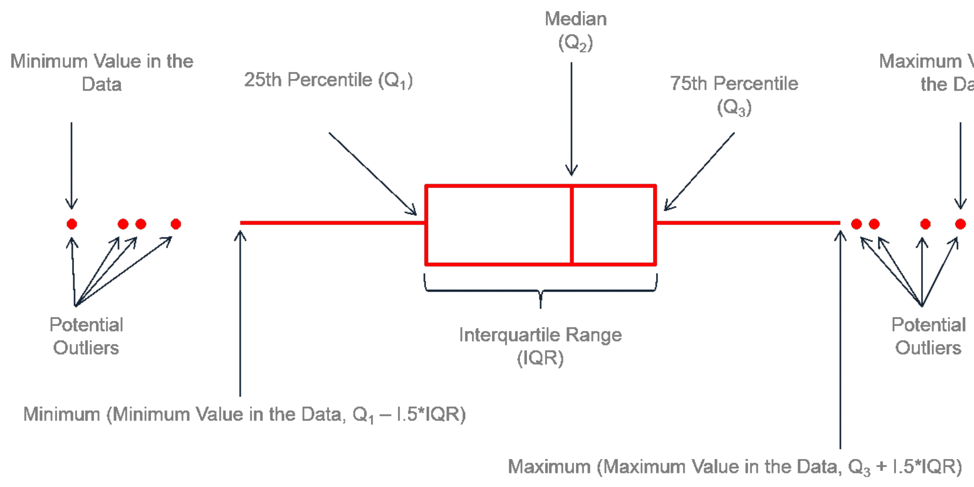
Five values are essential for constructing a boxplot:

- the median: the center of the data, middle value of a sorted list, 50% quantile of the data
- the first quantile (Q1): 25% quantile of the data
- the third quantile (Q3): 75% quantile of the data
- the interquantile range (IQR): the distance between Q1 and Q3

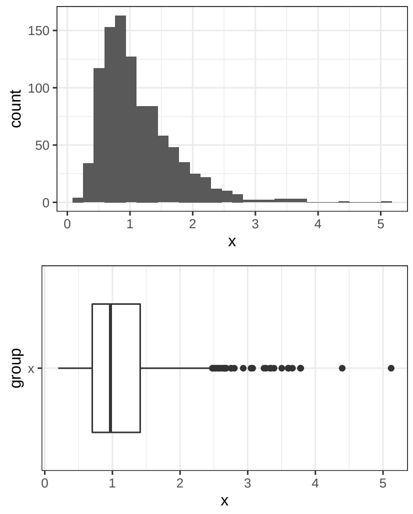
Every boxplot has lines at Q1, the median, and Q3, which together build the box of the boxplot. The other major feature of a boxplot is its whiskers. The whiskers are determined with the help of the IQR. Here, we compute 1.5 × IQR below Q1 and 1.5 × IQR above Q3. Anything outside of this range is called an outlier. We then draw lines at the smallest and largest point within this subset (Q1 - 1.5 × IQR to Q3 + 1.5 × IQR) from the dataset. These lines define our whiskers which reach the most extreme data point within $\pm 1.5 \times IQR$.



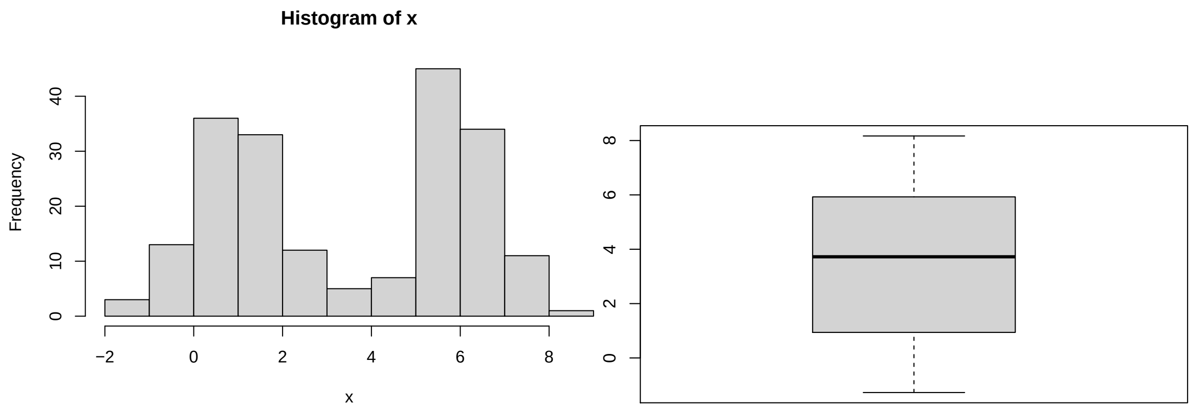
It is possible to not show the outliers in boxplots, as seen in the visualization before. However, we strongly recommend keeping them. Outliers can reveal interesting data points (discoveries “out of the box”) or bugs in data preprocessing.


Median (Q_2)
Minimum Value in the Data
25th Percentile (Q_1)
75th Percentile (Q_3)
Maximum Value in the Data
Potential Outliers
Interquartile Range (IQR)
Potential Outliers
Minimum (Minimum Value in the Data, Q_1 – 1.5*IQR)
Maximum (Maximum Value in the Data, Q_3 + 1.5*IQR)

For instance, we can plot the distribution of a variable `x` with a histogram and visualize the corresponding boxplot:



Boxplots are particularly suited for plotting non-gaussian symmetric and non-symmetric data and for plotting exponentially distributed data. However, boxplots are not well suited for bimodal data, since they only show one mode (the median). In the following example, we see a bimodal distribution in the histogram and the corresponding boxplot, which does not properly represent the distribution of the data.



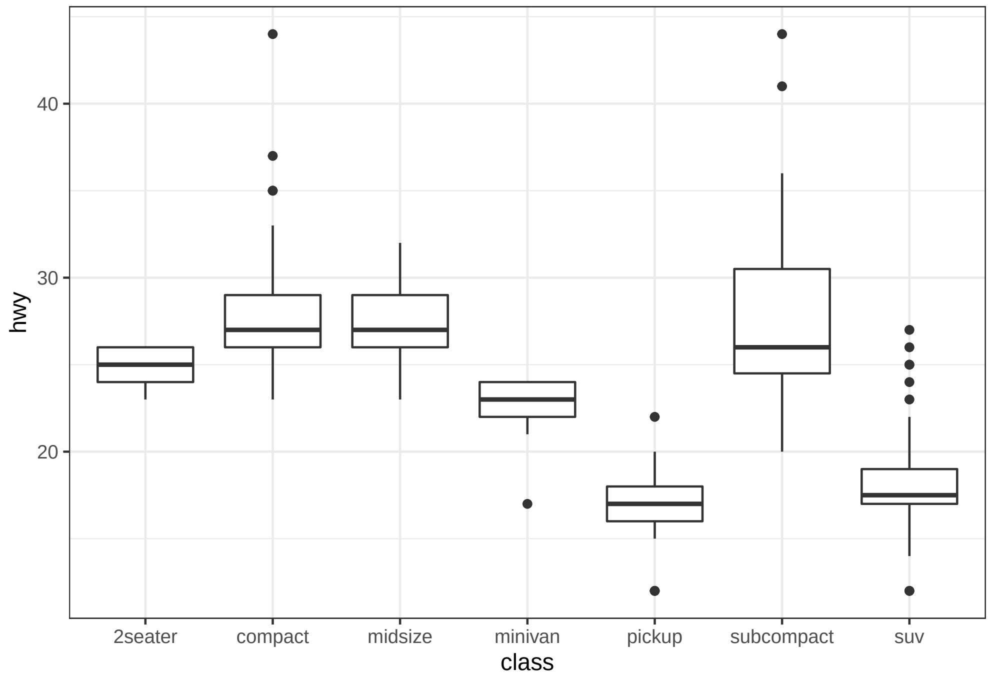
6.3.2 Plots for two variables: one continuous, one discrete

6.3.2.1 Boxplots by category

As illustrated before, boxplots are well suited for plotting one continuous variable. However, we can also use boxplots to show distributions of continuous variables with respect to some categories. This can be particularly interesting for comparing the different distributions of each category.

For instance, we want to visualize the highway miles per gallon `hwy` for every one of the 7 vehicle classes (compact, SUV, minivan, etc.). For this, we define the categorical `class` variable on the `x` axis and the continuous variable `hwy` on the `y` axis.

```
ggplot(mpg, aes(class, hwy)) + geom_boxplot() + mytheme
```

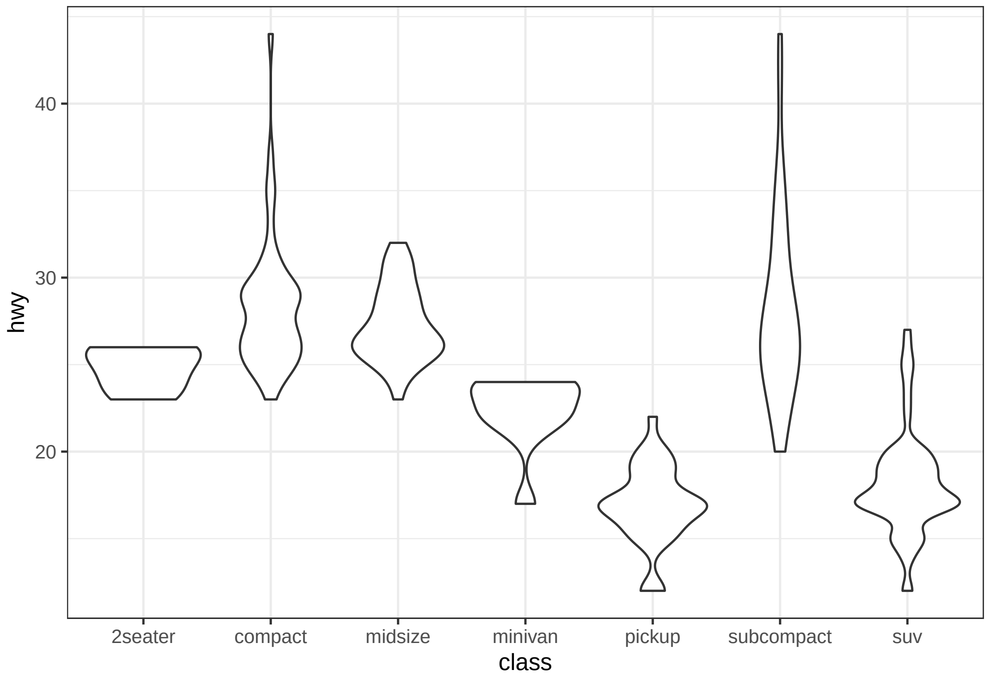


6.3.2.2 Violin plots

A violin plot is an alternative to the boxplot for visualizing either one continuous variable (grouped by categories). An advantage of the violin plot over the boxplot is that it also shows the entire distribution of the data. This can be particularly interesting when dealing with multimodal data.

For a direct comparison, we show a violin plot for the `hwy` grouped by `class` as before with the help of the function `geom_violin()`:

```
ggplot(mpg, aes(class, hwy)) + geom_violin() + mytheme
```

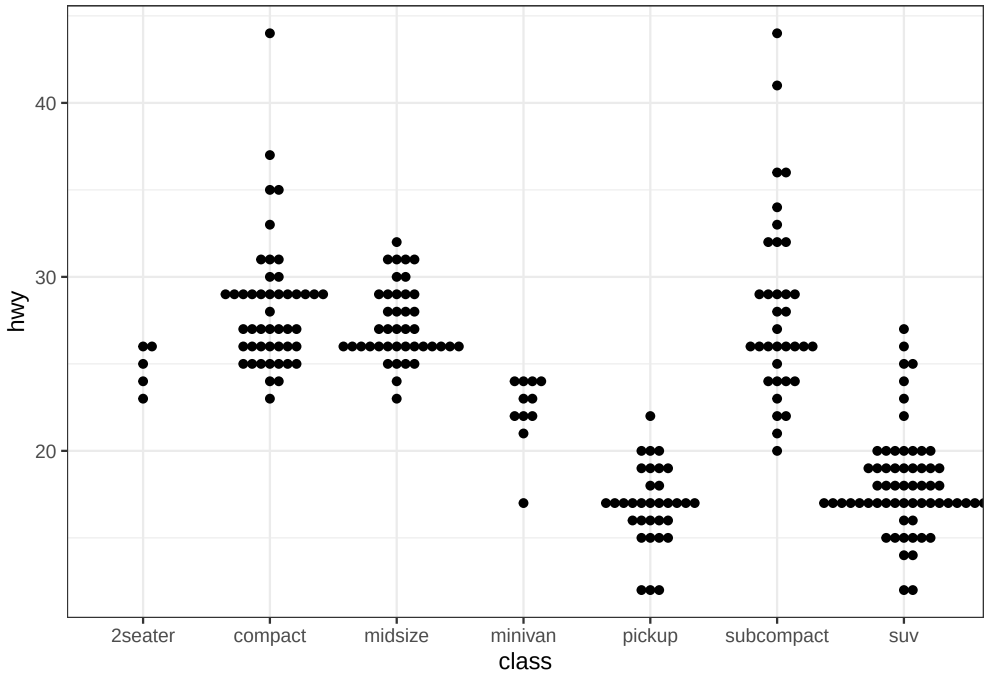


6.3.2.3 Beanplots

Another alternative to the popular boxplot is the beanplot. In a beanplot, the individual observations are shown as small lines in a one-dimensional scatter plot. Moreover, the estimated density of the distributions is visible in a beanplot. It is easy to compare different groups of data in a beanplot and to see if a group contains enough observations to make the group interesting from a statistical point of view.

For creating beanplots in R, we can use the package `ggbeeswarm`. We use the function `geom_besswarm()` to create a beanplot to visualize once again the `hwy` grouped by `class`:

```r
# install.packages('ggbeeswarm')
library(ggbeeswarm)
ggplot(mpg, aes(class, hwy)) + geom_beeswarm() + mytheme
```



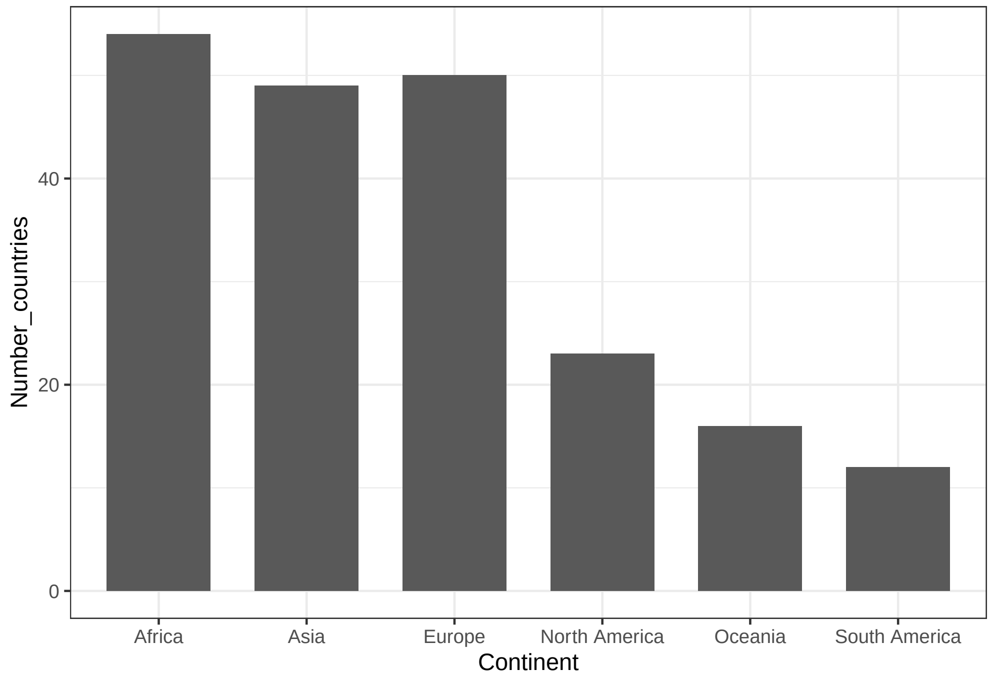
We remark that beanplots are useful only up to a certain number of data points. The creation of beanplots for larger datasets may become too expensive.

6.3.2.4 Barplots

Barplots are often used to highlight individual quantitative values per category. Bars are visual heavyweights compared to dots and lines. In a barplot, we can combine two attributes of 2-D location and line length to encode quantitative values. In this manner, we can focus the attention primarily on individual values and support the comparison of one to another.

For creating a barplot with `ggplot2` we can use the function `geom_bar()`. In the next example, we visualize the number of countries (defined in the `y` axis) per continent (defined in the `x` axis).

```r
ggplot(countries_dt, aes(Continent, Number_countries)) + geom_bar(stat = "identity",
    width = 0.7) + mytheme
```



6.3.2.5 Barplots with errorbars

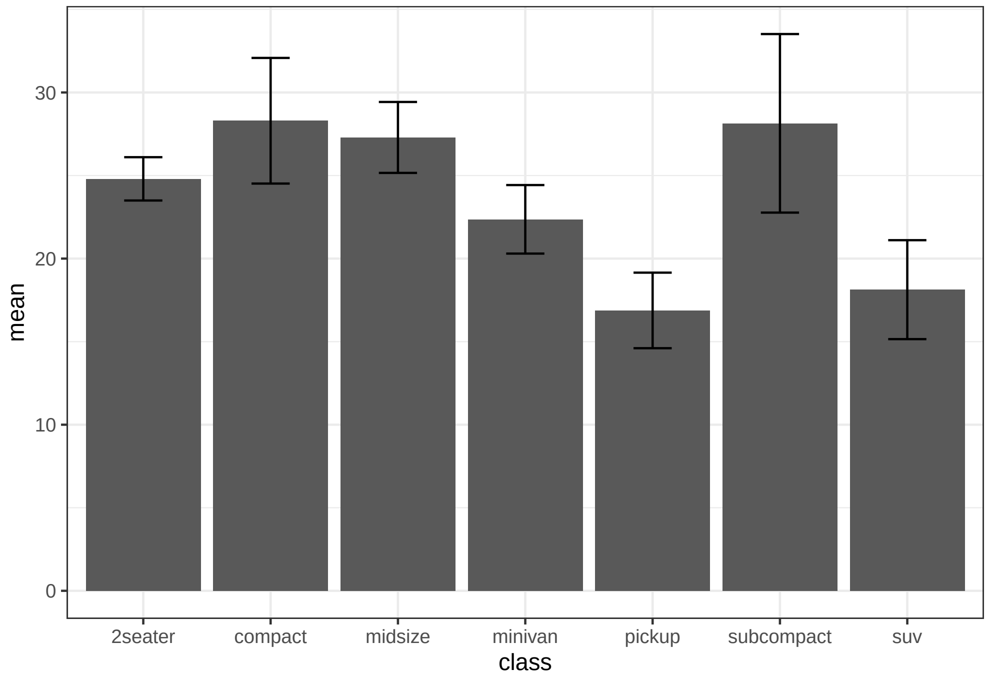
Visualizing uncertainty is important, otherwise, barplots with bars as a result of an aggregation can be misleading. One way to visualize uncertainty is with error bars.

As error bars, we can consider the standard deviation (SD) and the standard error of the mean (SEM). We remark that SD and SEM are completely different concepts. On the one hand, SD indicates the variation of quantity in the sample. On the other hand, SEM represents how well the mean is estimated.

The central limit theorem implies that: $SEM = SD/\sqrt{(n)}$, where n is the sample size (number of observations). With large n, SEM tends to 0.

In the following example, we plot me average highway miles per gallon `hwy` per vehicle class `class` including error bars computed as the average plus/minus standard devation of `hwy`:

```r
as.data.table(mpg) %>% .[, .(mean = mean(hwy), sd = sd(hwy)), by = class] %>% ggplot(aes(class,
    mean, ymax = mean + sd, ymin = mean - sd)) + geom_bar(stat = "identity") + geom_errorbar(width = 0.3)
    mytheme
```



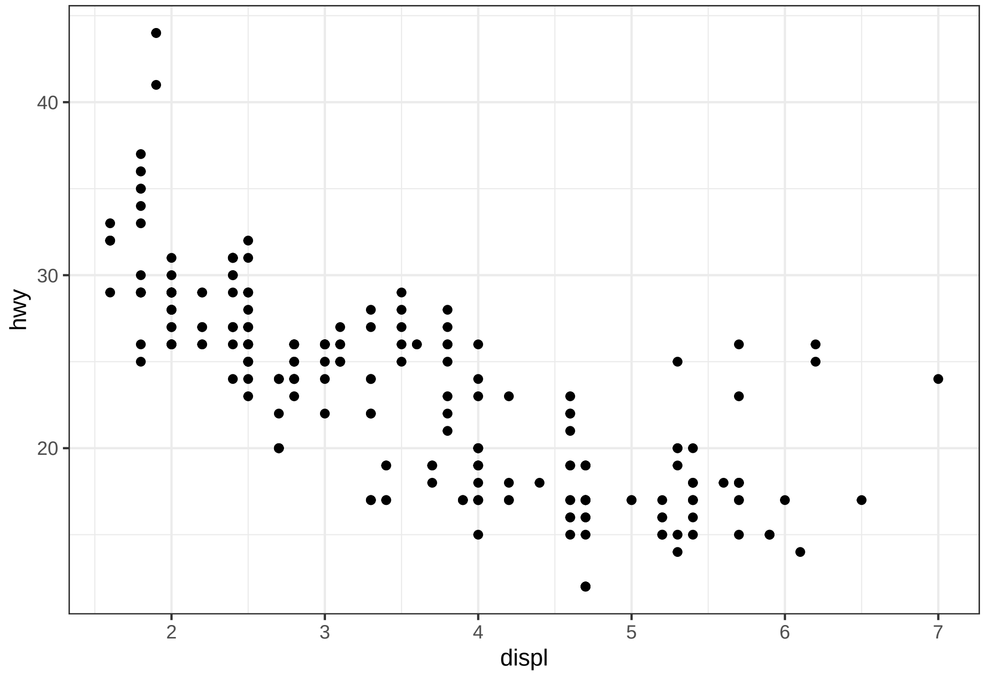
6.3.3 Plots for two continuos variables

6.3.3.1 Scatter plots

Scatter plots are a useful plot type for easily visualizing the relationship between two continuous variables. Here, dots are used to represent pairs of values corresponding to the two considered variables. The position of each dot on the horizontal (`x`) and vertical (`y`) axis indicates values for an individual data point.

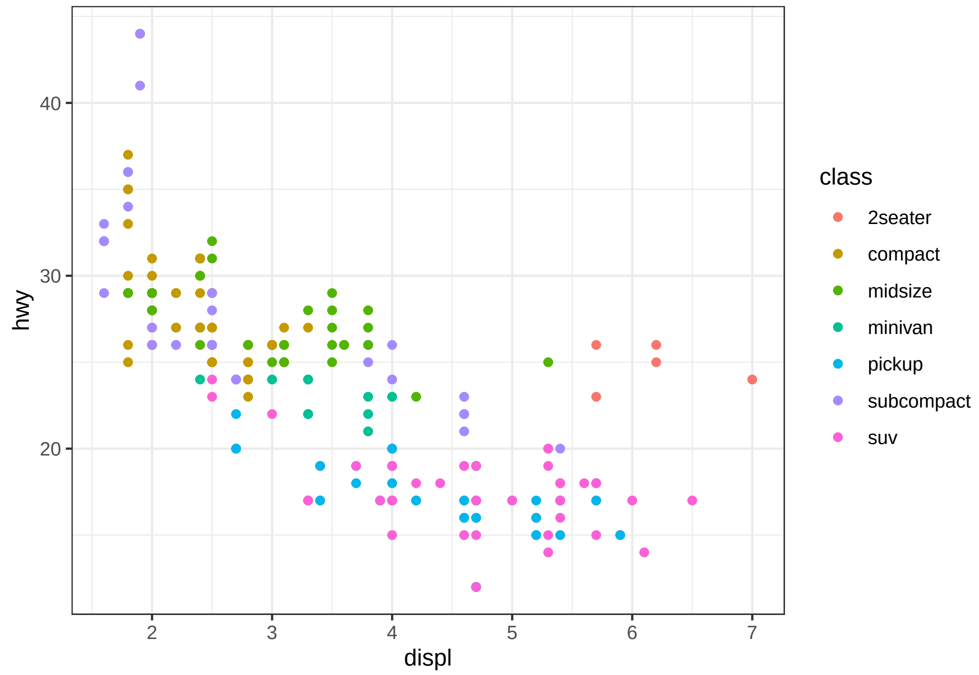
In the next example, we analyze the relationship between the engine displacement in liters `displ` and the highway miles per gallon `hwy` from the `mpg` dataset:

```
ggplot(mpg, aes(displ, hwy)) + geom_point() + mytheme
```



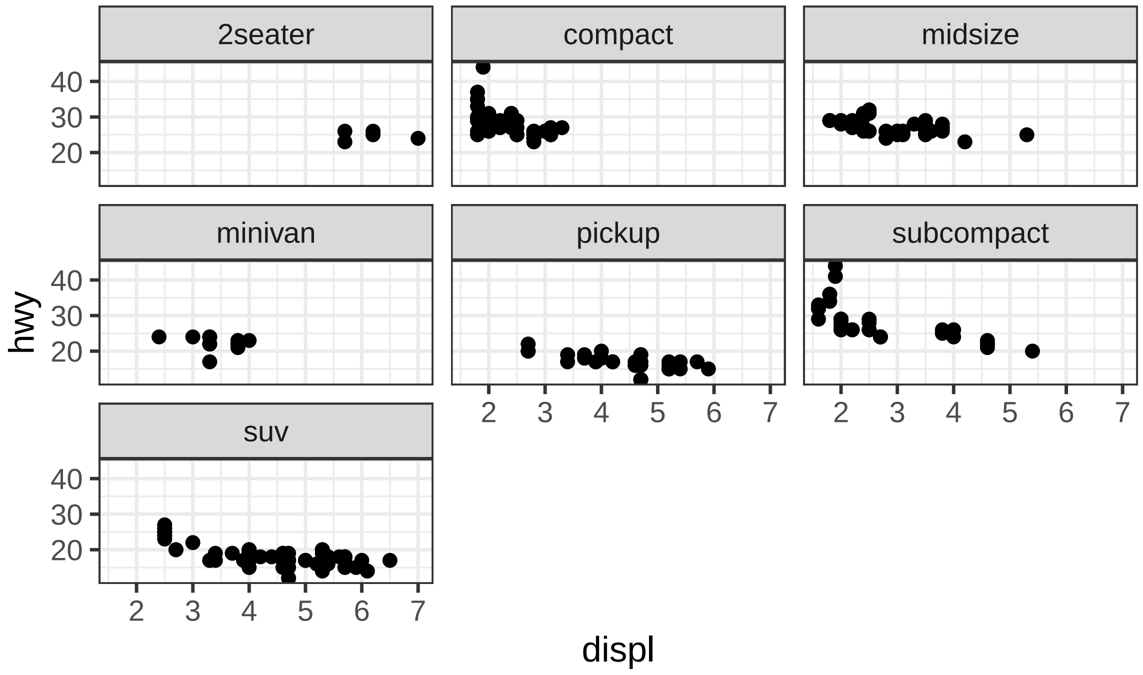
We can modify the previous plot by coloring the points depending on the vehicle class:

```
ggplot(mpg, aes(displ, hwy, color = class)) + geom_point() + mytheme
```



Sometimes, too many colors can be hard to distinguish. In such cases, we can use `facet` to separate them into different plots:

```
ggplot(mpg, aes(displ, hwy)) + geom_point() + facet_wrap(~class) + mytheme
```

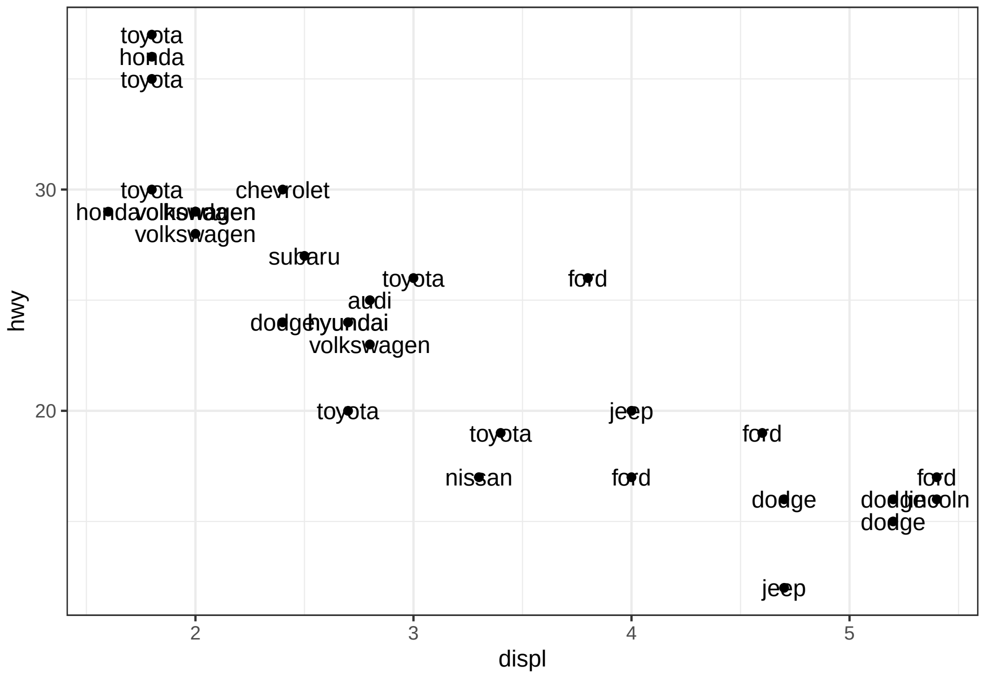


6.3.3.1.1 Text labeling

For labeling the individual points in a scatter plot, `ggplot2` offers the function `geom_text()`. However, these labels tend to overlap. To avoid this, we can use the library `ggrepel` which offers a better text labeling through the function `geom_text_repel()`.

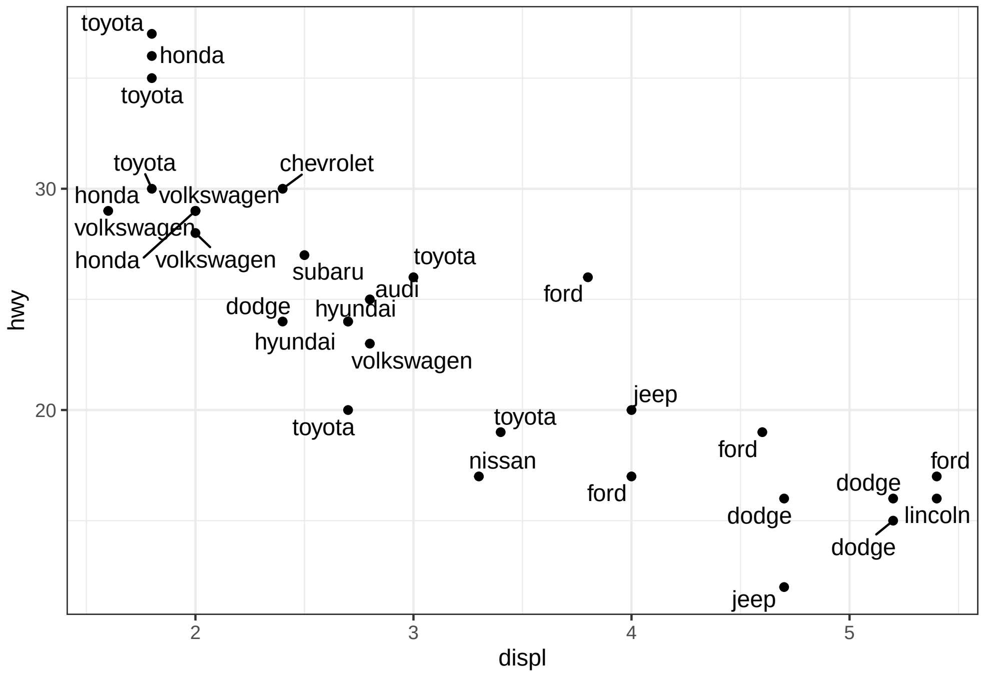
We first show the output of the classic text labeling with `geom_text()` for a random subset of 40 observations of the dataset `mpg`. Here we plot the engine displacement in liters `displ` vs. the highway miles per gallon `hwy` and label by `manufacturer`:

```
set.seed(12)
mpg_subset <- mpg[sample(1:nrow(mpg), 30, replace = FALSE), ]
ggplot(mpg_subset, aes(displ, hwy, label = manufacturer)) + geom_point() + geom_text() +
    mytheme
```



As seen in the previous illustration, the text labels overlap. This complicates understanding the plot. Therefore, we exchange the function `geom_text()` by `geom_text_repel()` from the library `ggrepel`:

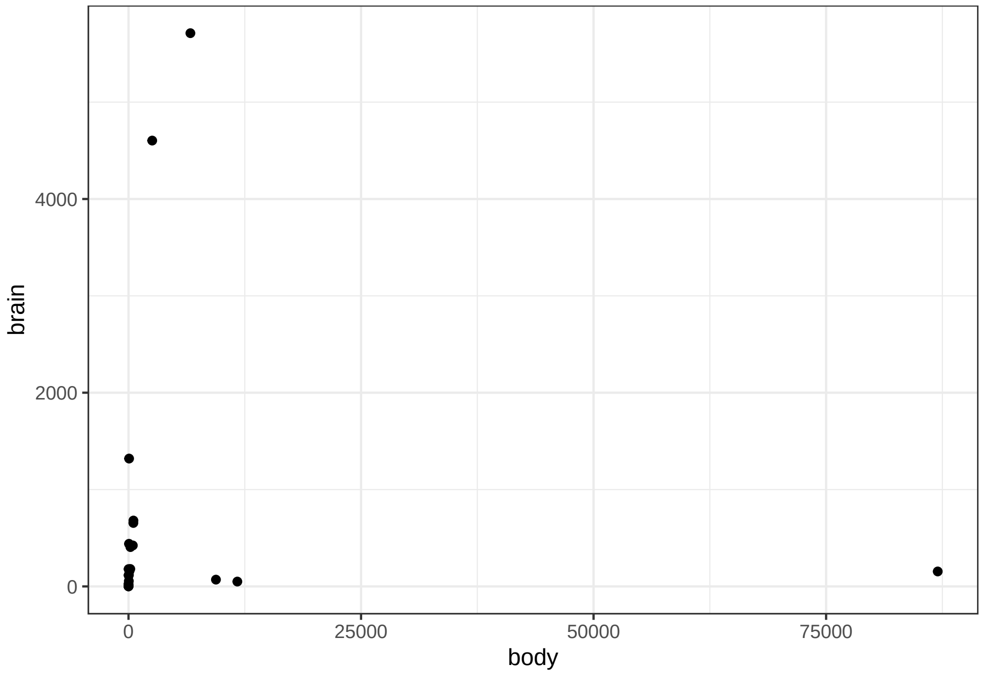
```
library(ggrepel)

ggplot(mpg_subset, aes(displ, hwy, label = manufacturer)) + geom_point() + geom_text_repel() +
    mytheme
```



6.3.3.1.2 Log scaling

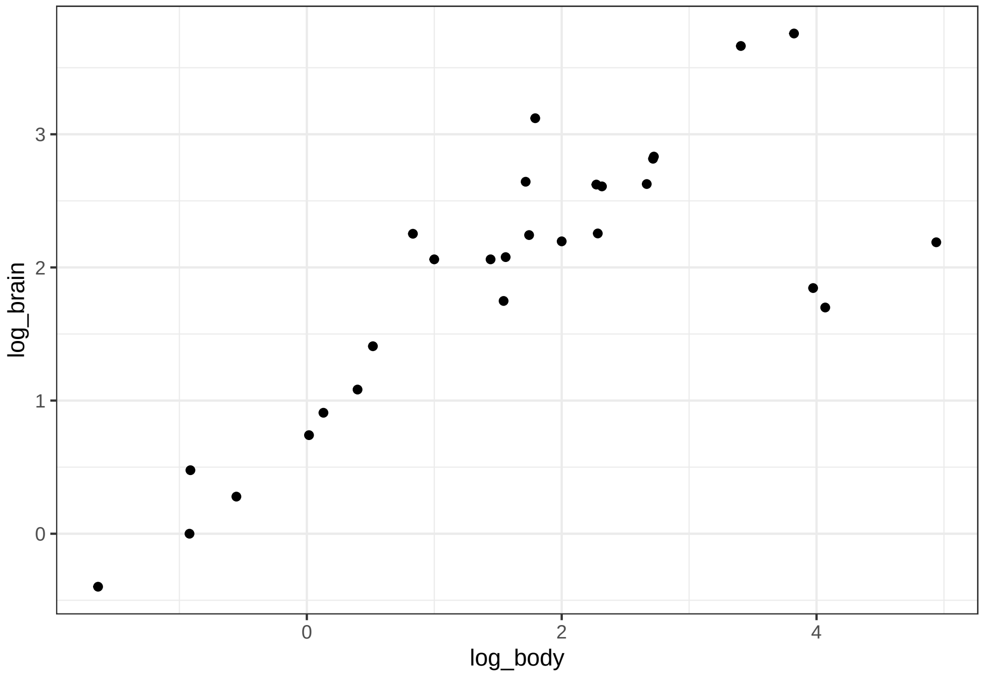
We consider another example where we want to plot the weights of the brain and body of different animals using the dataset `Animals`. This is what we obtain after creating a scatterplot.

```r
library(MASS)  # to access Animals data sets
animals_dt <- as.data.table(Animals)

ggplot(animals_dt, aes(x = body, y = brain)) + geom_point() + mytheme
```



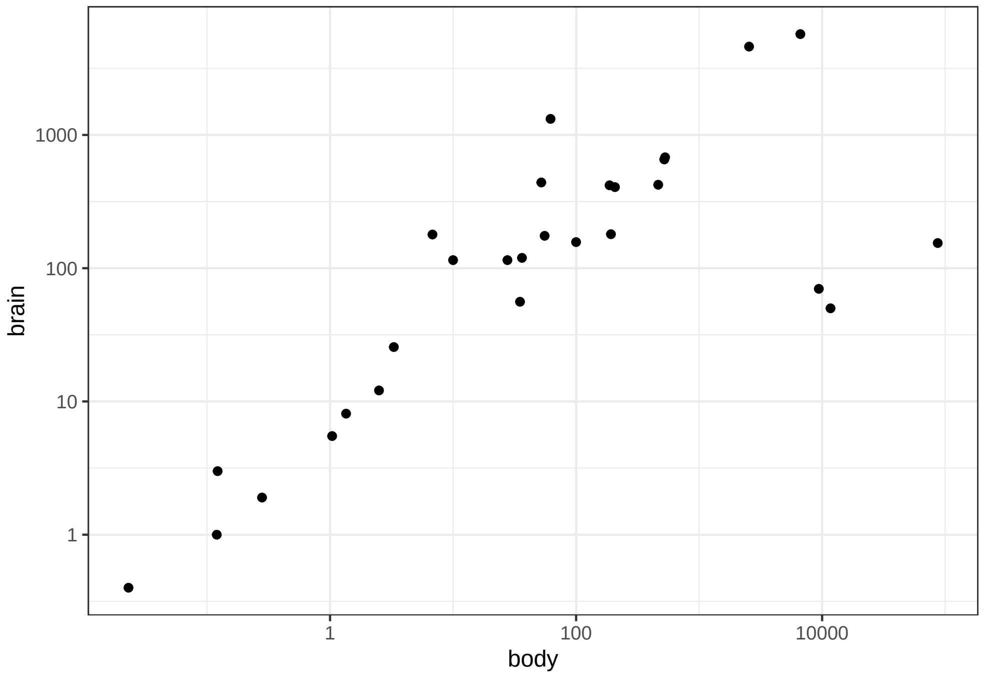
We can clearly see that there are a few points which are notably larger than most of the points. This makes it harder to interpret the relationships between most of these points. In such cases, we can consider **logarithmic** transformations and/or scaling. More precisely, a first idea would be to manually transform the values into a logarithmic space and plot the transformed values instead of the original values:

```
animals_dt[, `:=`(c("log_body", "log_brain"), list(log10(body), log10(brain)))]

ggplot(animals_dt, aes(x = log_body, y = log_brain)) + geom_point() + mytheme
```



Alternativelt, `ggplot2` offers to simply scale the data without the need to transform. This can be done with the help of the functions `scale_x_log10()` and `scale_y_log10()` which allow appropriate scaling and labeling of the axes:

```
ggplot(animals_dt, aes(x = body, y = brain)) + geom_point() + scale_x_log10() + scale_y_log10() +
    mytheme
```

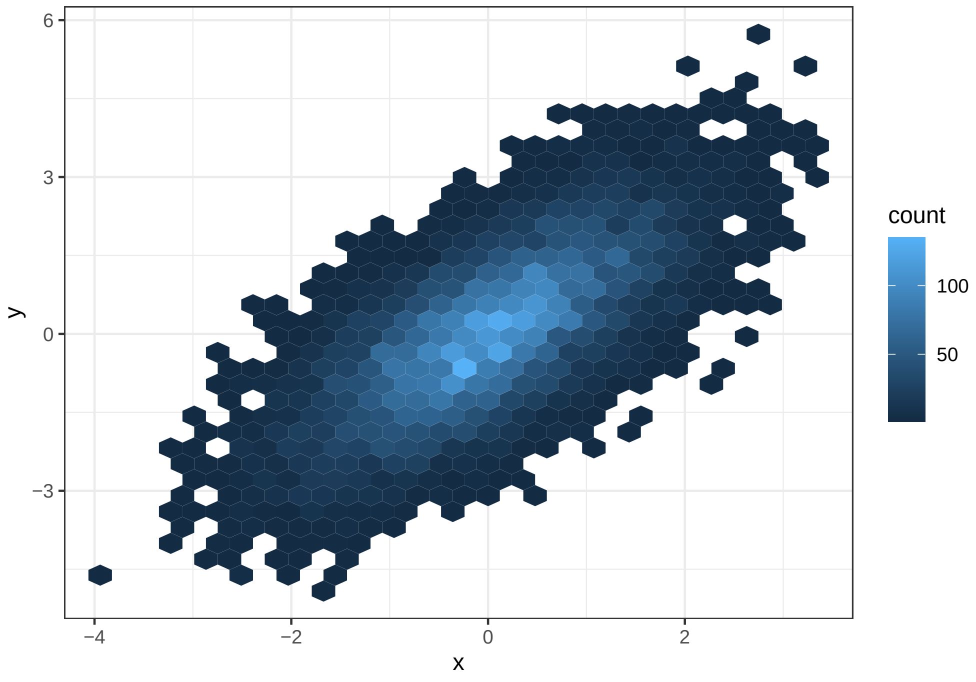


6.3.3.2 Density plots in 2D

Using scatterplots can become problematic when dealing with a huge number of points. This is due to the fact that points may overlap and we cannot clearly see how many points are at a certain position. In such cases, a 2D density plot is particularly well suited. This plot counts the number of observations within a particular area of the 2D space.

The function `geom_hex()` is particularly useful for creating 2D density plots in R:

```
x <- rnorm(10000)
y = x + rnorm(10000)
data.table(x, y) %>% ggplot(aes(x, y)) + geom_hex() + mytheme
```

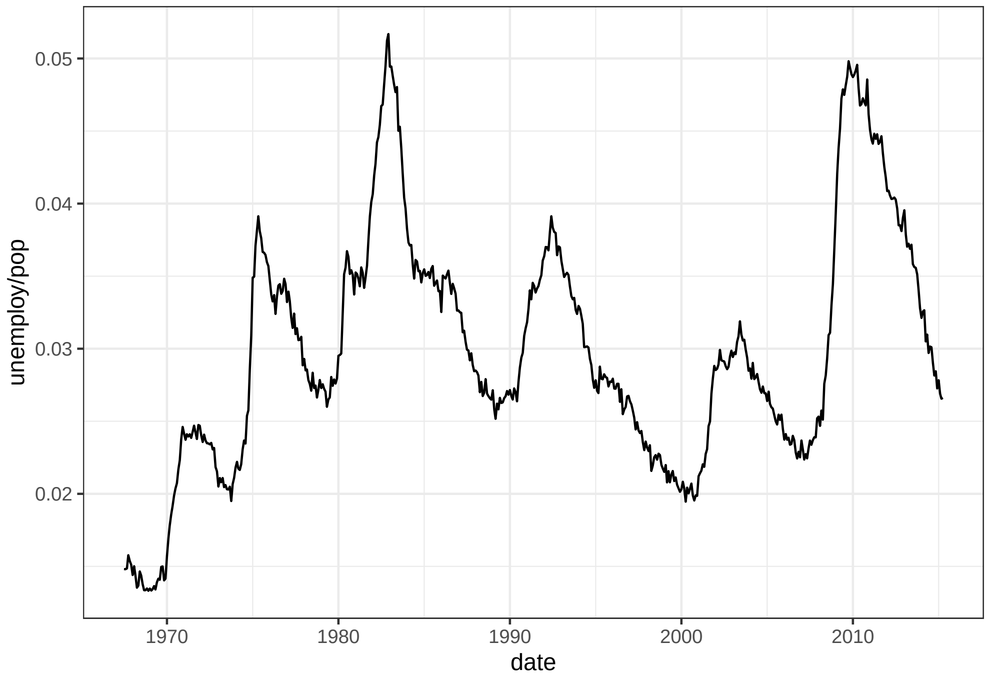


6.3.3.3 Line plots

A line plot can be considered for connecting a series of individual data points or to display the trend of a series of data points. This can be particularly useful to show the shape of data as it flows and changes from point to point. We can also show the strength of the movement of values up and down through time.

As an example we show the connection between the individual datapoints of unemployment rate over the years:

```
ggplot(economics, aes(date, unemploy/pop)) + geom_line() + mytheme
```



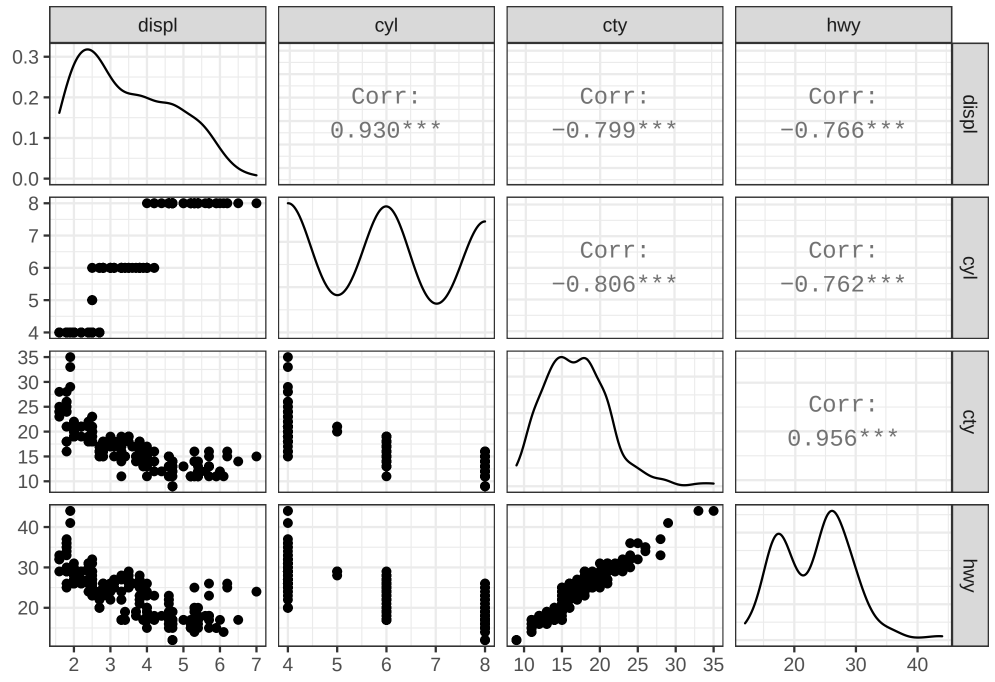
6.4 Further plots for low dimensional data

6.4.1 Plot matrix

A so-termed plot matrix is useful for exploring the distributions and correlations of a few variables in a matrix-like representation. Here, for each pair of considered variables, a scatterplot is created and the correlation coefficient is computed. For every single variable, a histogram is created for showing the respective distribution.

We can use the function `ggpairs()` from the library `GGally` for constructing plot matrices. As an example, we analyze the variables `displ`, `cyl`, `cty` and `hwy`from the dataset `mpg`:

```
library(GGally)
ggpairs(mpg, columns = c("displ", "cyl", "cty", "hwy")) + mytheme
```



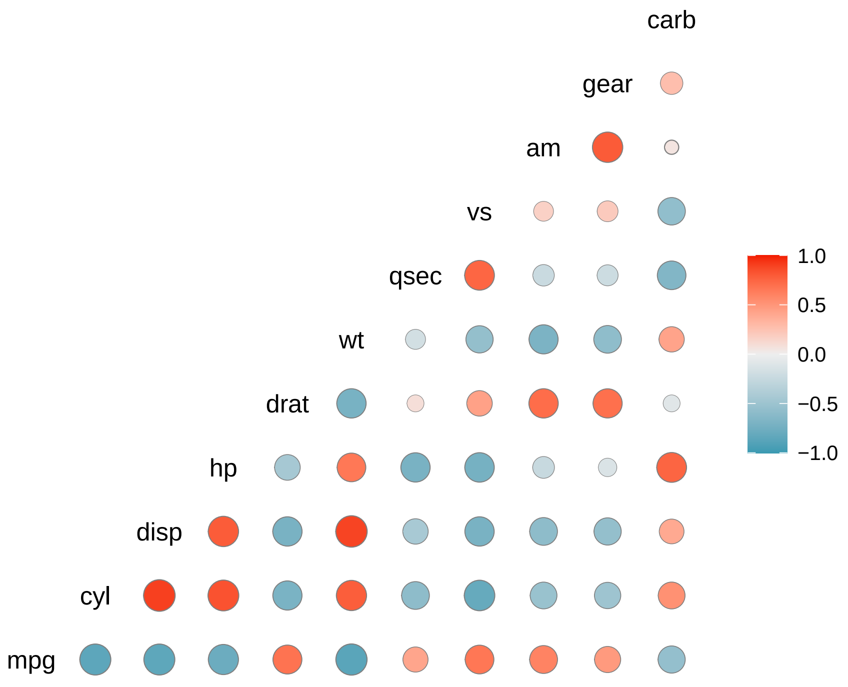
We remark that this plot is not well suited for comparing more than a few variables.

6.4.2 Correlation plot

A correlation plot is a graphical representation of a correlation matrix. It is useful to highlight the most correlated variables in a dataset. In this plot, correlation coefficients are colored according to the value. A correlation matrix can be also reordered according to the degree of association between variables.

Correlation plots are also called “corrgrams” or “correlograms”. As an example, we visualize the correlation between the variables of the dataset `mtcars` with the help of the function `ggcorr()` from the library `GGally`:

```
ggcorr(mtcars, geom = "circle")
```



6.5 Summary

This first chapter of data visualization covered the basics of the grammar of graphics and `ggplot2` to plot low dimensional data. We introduced the different types of plots such as histograms, boxplots and barplots and discussed when to use which plot.

6.6 Resources

- Plotting libraries:
  - http://www.r-graph-gallery.com/portfolio/ggplot2-package/
  - http://ggplot2.tidyverse.org/reference/
  - https://plot.ly/r/
  - https://plot.ly/ggplot2/
- Udacity's Data Visualization and D3.js
  - https://www.udacity.com/courses/all
- Graphics principles
  - https://onlinelibrary.wiley.com/doi/full/10.1002/pst.1912
  - https://graphicsprinciples.github.io/

Chapter 7

Reproducible data analysis with Snakemake

Basic idea:

- Decompose workflow into “rules” (steps).
- Rules define how to obtain output files from input files.
- Snakemake infers dependencies and execution order with a Directed Acyclic Graph.
- Any change in the input, only affected downstream rules will be executed.
- Disjoint paths in the DAG of jobs can be executed in parallel.
- Re-run all workflows with single `snakemake` command.

Example

```
rule sort:
    input:
        "path/to/dataset.txt"
    output:
        "dataset.sorted.txt"
    shell:
        "sort {input} > {output}"
```

```
rule sort:
    input:
        a="path/to/{dataset}.txt"
    output:
        b="{dataset}.sorted.txt"
    run:
        with open(output.b, "w") as out:
            for l in sorted(open(input.a)):
                print(l, file=out)
```

```
DATASETS = ["D1", "D2", "D3"] # use native python code

rule all:
    input:
        expand("{dataset}.sorted.txt", dataset=DATASETS)

rule sort:
    input:
        "path/to/{dataset}.txt"
    output:
        "{dataset}.sorted.txt"
    shell:
        "sort {input} > {output}"
```

Chapter 8

Quizes

8.1 data.table

1. From DT, display a named vector with the means of *y* and *v*. The names of the elements are mean_y and mean_v. Hint: remember which command returns a vector and which one a data.table.

   A. DT[, c(mean_y = mean(y), mean_v = mean(v))]

   B. DT[, list(mean_y = mean(y), mean_v = mean(v))]

   C. DT[, .(mean_y = mean(y), mean_v = mean(v))]

2. As part of a new project, you got the results of a set of experiments from a lab in a data.table (x) with 3 columns: experiment id, sample id and value. On each experiment, more than one sample was measured. On another data.table (y) you got the dates of all the experiments the lab has made after a certain date. Some experiments are not part of your project, but the lab did not subset the table. Before a certain date, the lab could not find the experiments date, but you don't want to discard those results. You want to merge both data tables in order to have only one with 4 columns: experiment id, sample id, value and experiment date. Which merge would you use? Hint: Sketch both data tables if necessary.

   A. Inner, all = FALSE

   B. Full, all = TRUE

   C. Left, all.x = TRUE

   D. Right, all.y = TRUE

8.2 tidy data

1. What transformation are required to tidy following data?

religion	<$10k	$10-20k	$20-30k	$30-40k	$40-50k	$50-75k
Agnostic	27	34	60	81	76	137
Atheist	12	27	37	52	35	70
Buddhist	27	21	30	34	33	58
Catholic	418	617	732	670	638	1116
Don't know/refused	15	14	15	11	10	35
Evangelical Prot	575	869	1064	982	881	1486
Hindu	1	9	7	9	11	34
Historically Black Prot	228	244	236	238	197	223
Jehovah's Witness	20	27	24	24	21	30
Jewish	19	19	25	25	30	95

A. Cast

B. Melt

C. Cast and Melt

D. Data are tidy already

2. What transformation are required to tidy following data?

id	year	month	element	d1	d2	d3	d4	d5	d6	d7	d8
MX17004	2010	1	tmax	—	—	—	—	—	—	—	—
MX17004	2010	1	tmin	—	—	—	—	—	—	—	—
MX17004	2010	2	tmax	—	27.3	24.1	—	—	—	—	—
MX17004	2010	2	tmin	—	14.4	14.4	—	—	—	—	—
MX17004	2010	3	tmax	—	—	—	—	32.1	—	—	—
MX17004	2010	3	tmin	—	—	—	—	14.2	—	—	—
MX17004	2010	4	tmax	—	—	—	—	—	—	—	—
MX17004	2010	4	tmin	—	—	—	—	—	—	—	—
MX17004	2010	5	tmax	—	—	—	—	—	—	—	—
MX17004	2010	5	tmin	—	—	—	—	—	—	—	—

A. Gather

B. Unite

C. Melt and cast

D. Melt, unite and cast

8.3 ggplot2

1. What's the result of the following command? `ggplot(data = mpg)`. Hint: ggplot builds plot layer by layer.

A. Nothing happens

B. A blank figure will be produced

C. A blank figure with axis will be produced

D. All data in `mpg` will be visualized

2. What's the result of the following command: `ggplot(data = mpg, aes(x = hwy, y = cty))`. Hint: ggplot builds plot layer by layer

   A. Nothing happens

   B. A blank figure will be produced

   C. A blank figure with axis will be produced

   D. A scatter plot will be produced

3. What's the result of the following command: `ggplot(data = mpg, aes(x = hwy, y = cty)) + geom_point()`. Hint: ggplot builds plot layer by layer.

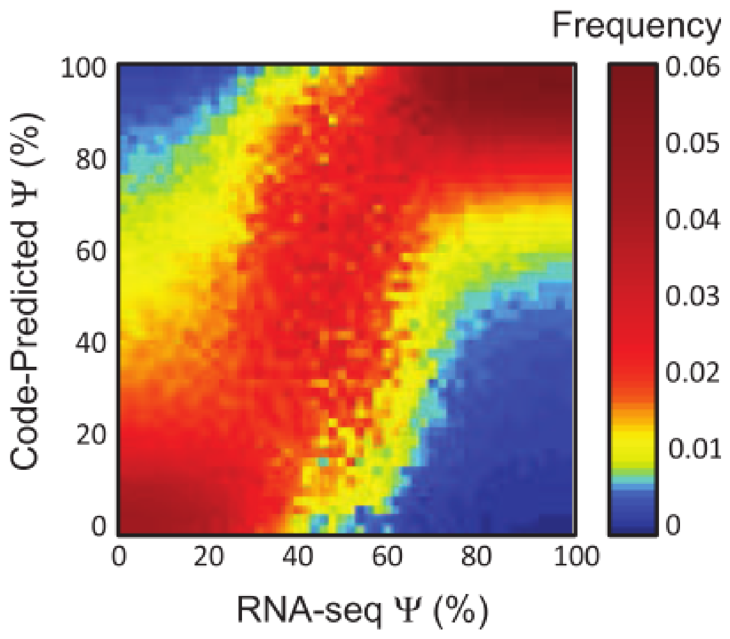
   A. Nothing happens

   B. A blank figure will be produced

   C. A blank figure with axis will be produced

   D. A scatter plot will be produced

4. What's wrong with the following plot?



Xiong et al., Science 2014

5. Which item is *not* chart junk?

   A. a bright red plot border

   B. light grey major grid lines

   C. bold labels and yellow grid lines

   D. data labels in Batik Gangster font

Chapter 9

Quiz solutions

9.1 data.table

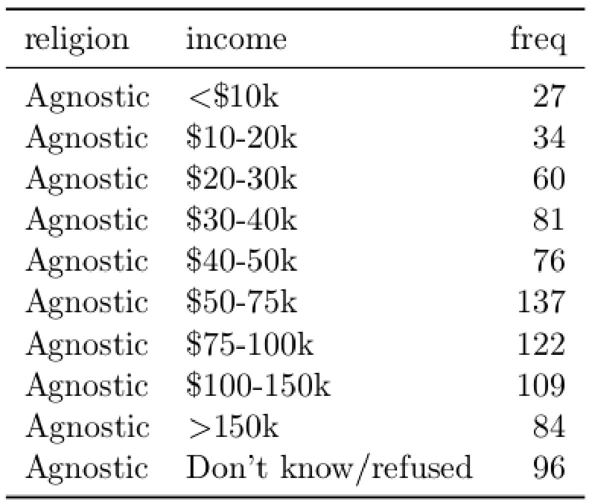
1. A, including the column names inside c() will return a vector.

2. C, inner join will omit the experiments whose dates are not provided. Full join will add rows of the experiments we are not interested in. Left join will leave the rows we are interested in and simply add the dates when available and NAs when not. Right join will leave us with experiments we are not interested and omit ones that we are interested but don't have the date.

9.2 tidy data

1. B, melt

   Tidy form:

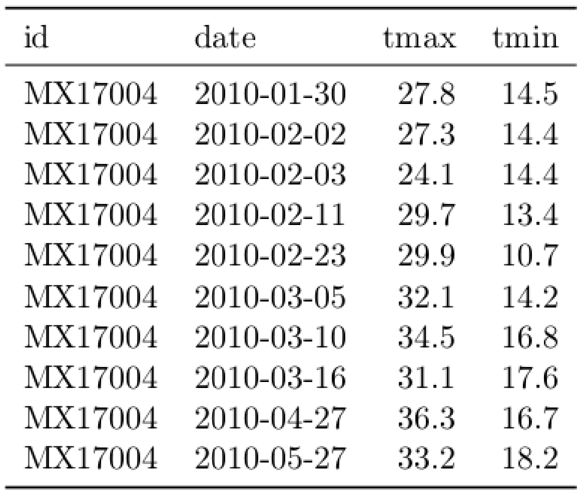


religion	income	freq
Agnostic	<$10k	27
Agnostic	$10-20k	34
Agnostic	$20-30k	60
Agnostic	$30-40k	81
Agnostic	$40-50k	76
Agnostic	$50-75k	137
Agnostic	$75-100k	122
Agnostic	$100-150k	109
Agnostic	>150k	84
Agnostic	Don't know/refused	96

Figure 9.1: Tidy religion dataset

2. D Melt, unite and cast

   Tidy form:



id	date	tmax	tmin
MX17004	2010-01-30	27.8	14.5
MX17004	2010-02-02	27.3	14.4
MX17004	2010-02-03	24.1	14.4
MX17004	2010-02-11	29.7	13.4
MX17004	2010-02-23	29.9	10.7
MX17004	2010-03-05	32.1	14.2
MX17004	2010-03-10	34.5	16.8
MX17004	2010-03-16	31.1	17.6
MX17004	2010-04-27	36.3	16.7
MX17004	2010-05-27	33.2	18.2

Figure 9.2: Tidy weather dataset

9.3 ggplot2

1. B, because neither variables were mapped nor geometry specified.

2. C, because while axis x and y are mapped, no geometry is specified.

3. D Axis x and y are mapped. But no geometry specified.

4. 1. Rainbow color where there are no breakpoints. Color boundaries are created based on author's interest.
   2. Manually twisted color scale.

5. B, light grey does not draw attention and grid lines are less important than data.

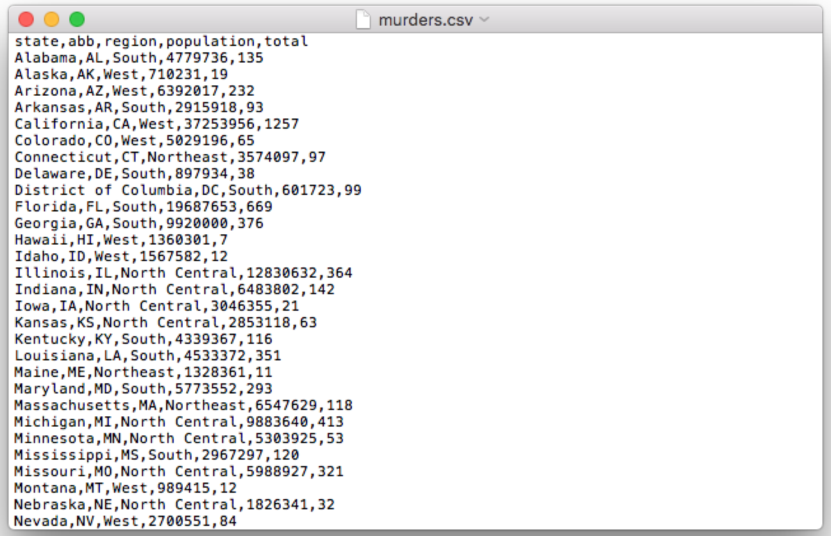
Chapter 10

Apendix I, Importing data

This chapter is largely adopted from “Introduction to Data Science” by Rafael A. Irizarry (https://rafalab.github.io/dsbook/) and covers the reading of files.

Usually the first step a data scientist has to do is to import data into R from either a file, a database, or other sources. Currently, one of the most common ways of storing and sharing data for analysis is through electronic spreadsheets. A spreadsheet stores data in rows and columns. It is basically a file version of a data frame. When saving such a table to a computer file, one needs a way to define when a new row or column ends and the other begins. This in turn defines the cells in which single values are stored.

When creating spreadsheets with text files, like the ones created with a simple text editor, a new row is defined with return and columns are separated with some predefined special character. The most common characters are comma (`,`), semicolon (`;`), space ( ), and tab (a preset number of spaces or `\t`). Here is an example of what a comma separated file looks like if we open it with a basic text editor (for example using the command `less` on Linux or Mac-OS systems):



```
state,abb,region,population,total
Alabama,AL,South,4779736,135
Alaska,AK,West,710231,19
Arizona,AZ,West,6392017,232
Arkansas,AR,South,2915918,93
California,CA,West,37253956,1257
Colorado,CO,West,5029196,65
Connecticut,CT,Northeast,3574097,97
Delaware,DE,South,897934,38
District of Columbia,DC,South,601723,99
Florida,FL,South,19687653,669
Georgia,GA,South,9920000,376
Hawaii,HI,West,1360301,7
Idaho,ID,West,1567582,12
Illinois,IL,North Central,12830632,364
Indiana,IN,North Central,6483802,142
Iowa,IA,North Central,3046355,21
Kansas,KS,North Central,2853118,63
Kentucky,KY,South,4339367,116
Louisiana,LA,South,4533372,351
Maine,ME,Northeast,1328361,11
Maryland,MD,South,5773552,293
Massachusetts,MA,Northeast,6547629,118
Michigan,MI,North Central,9883640,413
Minnesota,MN,North Central,5303925,53
Mississippi,MS,South,2967297,120
Missouri,MO,North Central,5988927,321
Montana,MT,West,989415,12
Nebraska,NE,North Central,1826341,32
Nevada,NV,West,2700551,84
```

The first row contains column names rather than data. We call this a *header*, and when we read-in data from a spreadsheet it is important to know if the file has a header or not. Most reading functions assume there is a header. To know if the file has a header, it helps to look at the file before trying to read it. This can be done with a text editor or with RStudio. In RStudio, we can do this by either opening the file in the editor or navigating to the file location, double clicking on the file, and hitting *View File*.

However, not all spreadsheet files are in a text format. Google Sheets, which are rendered on a browser, are an example. Another example is the proprietary format used by Microsoft Excel. These can't be viewed with a text editor. Despite this, due to the widespread use of Microsoft Excel software, this format is widely used.

We start this chapter by describing the difference between text (ASCII), Unicode, and binary files and how this affects how we import them. We then explain the concepts of file paths and working directories, which are essential to understand how to import data effectively. We then introduce the **readr** and **readxl** package and the functions that are available to import spreadsheets into R. Finally, we provide some recommendations on how to store and organize data in files. More complex challenges such as extracting data from web pages or PDF documents are left for the Data Wrangling part of the book.

10.1 Paths and the working directory

The first step when importing data from a spreadsheet is to locate the file containing the data. Although we do not recommend it, you can use an approach similar to what you do to open files in Microsoft Excel by clicking on the RStudio "File" menu, clicking "Import Dataset", then clicking through folders until you find the file. We want to be able to write code rather than use the point-and-click approach. The keys and concepts we need to learn to do this are described in detail in the Productivity Tools part of this book. Here we provide an overview of the very basics.

The main challenge in this first step is that we need to let the R functions doing the importing know where to look for the file containing the data. The simplest way to do this is to have a copy of the file in the folder in which the importing functions look by default. Once we do this, all we have to supply to the importing function is the filename.

A spreadsheet containing the US murders data is included as part of the **dslabs** package. Finding this file is not straightforward, but the following lines of code copy the file to the folder in which R looks in by default. We explain how these lines work below.

```r
filename <- "murders.csv"
dir <- system.file("extdata", package = "dslabs")
fullpath <- file.path(dir, filename)
file.copy(fullpath, "murders.csv")
```

This code does not read the data into R, it just copies a file. But once the file is copied, we can import the data with a simple line of code. Here we use the `fread` function from the **data.table** package.

```r
library(data.table)
dt <- fread(filename)
```

Here the data is read into a `data.table` called `dt`.

Currently fread is the fasted option and suitable for large data sets. However, there are alternative options like the `read_csv` function from the **readr** package, which is part of the tidyverse.

```r
library(tidyverse)
dat <- read_csv(filename)
```

The data is imported and stored in `dat`. The rest of this section defines some important concepts and provides an overview of how we write code that tells R how to find the files we want to import. Rafael A. Irizarry's Chapter on Productivity Tools covers this in depth.

10.1.1 The filesystem

You can think of your computer's filesystem as a series of nested folders, each containing other folders and files. Data scientists refer to folders as *directories*. We refer to the folder that contains all other folders as the *root directory*. We refer to the directory in which we are currently located as the *working directory*. The working directory therefore changes as you move through folders: think of it as your current location.

10.1.2 Relative and full paths

The *path* of a file is a list of directory names that can be thought of as instructions on what folders to click on, and in what order, to find the file. If these instructions are for finding the file from the root directory we refer to it as the *full path*. If the instructions are for finding the file starting in the working directory we refer to it as a *relative path*.

To see an example of a full path on your system type the following:

```r
system.file(package = "dslabs")
```

The strings separated by slashes are the directory names. The first slash represents the root directory and we know this is a full path because it starts with a slash. If the first directory name appears without a slash in front, then the path is assumed to be relative. We can use the function `list.files` to see examples of relative paths.

```r
dir <- system.file(package = "dslabs")
list.files(path = dir)
```

```
##  [1] "data"        "DESCRIPTION" "extdata"     "help"        "html"
##  [6] "INDEX"       "Meta"        "NAMESPACE"   "R"           "script"
```

These relative paths give us the location of the files or directories if we start in the directory with the full path. For example, the full path to the `help` directory in the example above is `/Library/Frameworks/R.framework/Versions/3.5/Resources`

Note: You will probably not make much use of the `system.file` function in your day-to-day data analysis work. We introduce it in this section because it facilitates the sharing of spreadsheets by including them in the **dslabs** package. You will rarely have the luxury of data being included in packages you already have installed. However, you will frequently need to navigate full and relative paths and import spreadsheet formatted data.

10.1.3 The working directory

We highly recommend only writing relative paths in your code. The reason is that full paths are unique to your computer and you want your code to be portable. You can get the full path of your working directory without writing out explicitly by using the `getwd` function.

```r
wd <- getwd()
```

If you need to change your working directory, you can use the function `setwd` or you can change it through RStudio by clicking on "Session".

10.1.4 Generating path names

Another example of obtaining a full path without writing out explicitly was given above when we created the object `fullpath` like this:

```r
filename <- "murders.csv"
dir <- system.file("extdata", package = "dslabs")
fullpath <- file.path(dir, filename)
```

The function `system.file` provides the full path of the folder containing all the files and directories relevant to the package specified by the `package` argument. By exploring the directories in `dir` we find that the `extdata` contains the file we want:

```r
dir <- system.file(package = "dslabs")
filename %in% list.files(file.path(dir, "extdata"))
```

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

The `system.file` function permits us to provide a subdirectory as a first argument, so we can obtain the fullpath of the `extdata` directory like this:

```
dir <- system.file("extdata", package = "dslabs")
```

The function `file.path` is used to combine directory names to produce the full path of the file we want to import.

```
fullpath <- file.path(dir, filename)
```

10.1.5 Copying files using paths

The final line of code we used to copy the file into our home directory used
the function `file.copy`. This function takes two arguments: the file to copy and the name to give it in the new directory.

```
file.copy(fullpath, "murders.csv")
```

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

If a file is copied successfully, the `file.copy` function returns `TRUE`. Note that we are giving the file the same name, `murders.csv`, but we could have named it anything. Also note that by not starting the string with a slash, R assumes this is a relative path and copies the file to the working directory.

You should be able to see the file in your working directory and can check by using:

```
list.files()
```

10.2 The readr and readxl packages

In this section we introduce the main tidyverse data importing functions. We will use the `murders.csv` file provided by the **dslabs** package as an example. To simplify the illustration we will copy the file to our working directory using the following code:

```
filename <- "murders.csv"
dir <- system.file("extdata", package = "dslabs")
fullpath <- file.path(dir, filename)
file.copy(fullpath, "murders.csv")
```

10.2.1 readr

The **readr** library includes functions for reading data stored in text file spreadsheets into R. **readr** is part of the **tidyverse** package, or you can load it directly:

```
library(readr)
```

The following functions are available to read-in spreadsheets:

Function	Format	Typical suffix
read_table	white space separated values	txt
read_csv	comma separated values	csv
read_csv2	semicolon separated values	csv
read_tsv	tab delimited separated values	tsv
read_delim	general text file format, must define delimiter	txt

Although the suffix usually tells us what type of file it is, there is no guarantee that these always match. We can open the file to take a look or use the function `read_lines` to look at a few lines:

```r
read_lines("murders.csv", n_max = 3)
```

```
## [1] "state,abb,region,population,total" "Alabama,AL,South,4779736,135"
## [3] "Alaska,AK,West,710231,19"
```

This also shows that there is a header. Now we are ready to read-in the data into R. From the .csv suffix and the peek at the file, we know to use `read_csv`:

```r
dat <- read_csv(filename)
```

Note that we receive a message letting us know what data types were used for each column. Also note that `dat` is a `tibble`, not just a data frame. This is because `read_csv` is a **tidyverse** parser. We can confirm that the data has in fact been read-in with:

```r
View(dat)
```

Finally, note that we can also use the full path for the file:

```r
dat <- read_csv(fullpath)
```

10.2.2 readxl

You can load the **readxl** package using

```r
library(readxl)
```

The package provides functions to read-in Microsoft Excel formats:

Function	Format	Typical suffix
read_excel	auto detect the format	xls, xlsx
read_xls	original format	xls
read_xlsx	new format	xlsx

The Microsoft Excel formats permit you to have more than one spreadsheet in one file. These are referred to as *sheets*. The functions listed above read the first sheet by default, but we can also read the others. The `excel_sheets` function gives us the names of all the sheets in an Excel file. These names can then be passed to the `sheet` argument in the three functions above to read sheets other than the first.

10.3 Exercises

1. Use the `read_csv` function to read each of the files that the following code saves in the `files` object:

```r
path <- system.file("extdata", package = "dslabs")
files <- list.files(path)
files
```

2. Note that the last one, the `olive` file, gives us a warning. This is because the first line of the file is missing the header for the first column.

Read the help file for `read_csv` to figure out how to read in the file without reading this header. If you skip the header, you should not get this warning. Save the result to an object called `dat`.

3. A problem with the previous approach is that we don't know what the columns represent. Type:

```r
names(dat)
```

to see that the names are not informative.

Use the `readLines` function to read in just the first line (we later learn how to extract values from the output).

10.4 Downloading files

Another common place for data to reside is on the internet. When these data are in files, we can download them and then import them or even read them directly from the web. For example, we note that because our **dslabs** package is on GitHub, the file we downloaded with the package has a url:

```
url <- "https://raw.githubusercontent.com/rafalab/dslabs/master/inst/
extdata/murders.csv"
```

The `read_csv` file can read these files directly:

```
dat <- read_csv(url)
```

If you want to have a local copy of the file, you can use the `download.file` function:

```
download.file(url, "murders.csv")
```

This will download the file and save it on your system with the name `murders.csv`. You can use any name here, not necessarily `murders.csv`. Note that when using `download.file` you should be careful as **it will overwrite existing files without warning**.

Two functions that are sometimes useful when downloading data from the internet are `tempdir` and `tempfile`. The first creates a directory with a random name that is very likely to be unique. Similarly, `tempfile` creates a character string, not a file, that is likely to be a unique filename. So you can run a command like this which erases the temporary file once it imports the data:

```
tmp_filename <- tempfile()
download.file(url, tmp_filename)
dat <- read_csv(tmp_filename)
file.remove(tmp_filename)
```

10.5 R-base importing functions

R-base also provides import functions. These have similar names to those in the **tidyverse**, for example `read.table`, `read.csv` and `read.delim`. However, there are a couple of important differences. To show this we read-in the data with an R-base function:

```
dat2 <- read.csv(filename)
```

An important difference is that the characters are converted to factors:

```
class(dat2$abb)
```

```
## [1] "character"
```

```
class(dat2$region)
```

```
## [1] "character"
```

This can be avoided by setting the argument `stringsAsFactors` to `FALSE`.

```
dat <- read.csv("murders.csv", stringsAsFactors = FALSE)
class(dat$state)
```

```
## [1] "character"
```

In our experience this can be a cause for confusion since a variable that was saved as characters in file is converted to factors regardless of what the variable represents. In fact, we **highly** recommend setting `stringsAsFactors=FALSE` to be your default approach when using the R-base parsers. You can easily convert the desired columns to factors after importing data.

10.5.1 `scan`

When reading in spreadsheets many things can go wrong. The file might have a multiline header, be missing cells, or it might use an unexpected encoding[1]. We recommend you read this post about common issues found here: https://www.joelonsoftware.com/2003/10/08/the-absolute-minimum-every-software-developer-absolutely-positively-must-know-about-unicode-and-character-sets-no-excuses/.

With experience you will learn how to deal with different challenges. Carefully reading the help files for the functions discussed here will be useful. With scan you can read-in each cell of a file. Here is an example:

```
path <- system.file("extdata", package = "dslabs")
filename <- "murders.csv"
x <- scan(file.path(path, filename), sep = ",", what = "c")
x[1:10]
```

```
##  [1] "state"      "abb"        "region"     "population" "total"
##  [6] "Alabama"    "AL"         "South"      "4779736"    "135"
```

Note that the tidyverse provides `read_lines`, a similarly useful function.

10.6 Text versus binary files

For data science purposes, files can generally be classified into two categories: text files (also known as ASCII files) and binary files. You have already worked with text files. All your R scripts are text files and so are the R markdown files used to create this book. The csv tables you have read are also text files. One big advantage of these files is that we can easily “look” at them without having to purchase any kind of special software or follow complicated instructions. Any text editor can be used to examine a text file, including freely available editors such as RStudio, Notepad, textEdit, vi, emacs, nano, and pico. To see this, try opening a csv file using the “Open file” RStudio tool. You should be able to see the content right on your editor. However, if you try to open, say, an Excel xls file, jpg or png file, you will not be able to see anything immediately useful. These are binary files. Excel files are actually compressed folders with several text files inside. But the main distinction here is that text files can be easily examined.

Although R includes tools for reading widely used binary files, such as xls files, in general you will want to find data sets stored in text files. Similarly, when sharing data you want to make it available as text files as long as storage is not an issue (binary files are much more efficient at saving space on your disk). In general, plain-text formats make it easier to share data since commercial software is not required for working with the data.

Extracting data from a spreadsheet stored as a text file is perhaps the easiest way to bring data from a file to an R session. Unfortunately, spreadsheets are not always available and the fact that you can look at text files does not necessarily imply that extracting data from them will be straightforward. In the Data Wrangling part of the book we learn to extract data from more complex text files such as html files.

10.7 Unicode versus ASCII

A pitfall in data science is assuming a file is an ASCII text file when, in fact, it is something else that can look a lot like an ASCII text file: a Unicode text file.

To understand the difference between these, remember that everything on a computer needs to eventually be converted to 0s and 1s. ASCII is an *encoding* that maps characters to numbers. ASCII uses 7 bits (0s and 1s) which

[1] https://en.wikipedia.org/wiki/Character_encoding

results in $2^7 = 128$ unique items, enough to encode all the characters on an English language keyboard. However, other languages use characters not included in this encoding. For example, the é in México is not encoded by ASCII. For this reason, a new encoding, using more than 7 bits, was defined: Unicode. When using Unicode, one can chose between 8, 16, and 32 bits abbreviated UTF-8, UTF-16, and UTF-32 respectively. RStudio actually defaults to UTF-8 encoding.

Although we do not go into the details of how to deal with the different encodings here, it is important that you know these different encodings exist so that you can better diagnose a problem if you encounter it. One way problems manifest themselves is when you see "weird looking" characters you were not expecting. This StackOverflow discussion is an example: https://stackoverflow.com/questions/18789330/r-on-windows-character-encoding-hell.

10.8 Organizing data with spreadsheets

Although this book focuses almost exclusively on data analysis, data management is also an important part of data science. As explained in the introduction, we do not cover this topic. However, quite often data analysts needs to collect data, or work with others collecting data, in a way that is most conveniently stored in a spreadsheet. Although filling out a spreadsheet by hand is a practice we highly discourage, we instead recommend the process be automatized as much as possible, sometimes you just have to do it. Therefore, in this section, we provide recommendations on how to organize data in a spreadsheet. Although there are R packages designed to read Microsoft Excel spreadsheets, we generally want to avoid this format. Instead, we recommend Google Sheets as a free software tool. Below we summarize the recommendations made in paper by Karl Broman and Kara Woo[2]. Please read the paper for important details.

- **Be Consistent** - Before you commence entering data, have a plan. Once you have a plan, be consistent and stick to it.
- **Choose Good Names for Things** - You want the names you pick for objects, files, and directories to be memorable, easy to spell, and descriptive. This is actually a hard balance to achieve and it does require time and thought. One important rule to follow is **do not use spaces**, use underscores _ or dashes instead -. Also, avoid symbols; stick to letters and numbers.
- **Write Dates as YYYY-MM-DD** - To avoid confusion, we strongly recommend using this global ISO 8601 standard.
- **No Empty Cells** - Fill in all cells and use some common code for missing data.
- **Put Just One Thing in a Cell** - It is better to add columns to store the extra information rather than having more than one piece of information in one cell.
- **Make It a Rectangle** - The spreadsheet should be a rectangle.
- **Create a Data Dictionary** - If you need to explain things, such as what the columns are or what the labels used for categorical variables are, do this in a separate file.
- **No Calculations in the Raw Data Files** - Excel permits you to perform calculations. Do not make this part of your spreadsheet. Code for calculations should be in a script.
- **Do Not Use Font Color or Highlighting as Data** - Most import functions are not able to import this information. Encode this information as a variable instead.
- **Make Backups** - Make regular backups of your data.
- **Use Data Validation to Avoid Errors** - Leverage the tools in your spreadsheet software so that the process is as error-free and repetitive-stress-injury-free as possible.
- **Save the Data as Text Files** - Save files for sharing in comma or tab delimited format.

[2]https://www.tandfonline.com/doi/abs/10.1080/00031305.2017.1375989

Chapter 11

Appendix II, R basics

This Appendix covers the basic concepts of the R programming language: data types, data operations, data structures, control flow and functions.

11.1 Data structures

R has various data structures and types that we will see shortly. From simple scalar variables to tables with multiple features and observation, each layer builds on top of its previous.

Notes:

- Scalars are just vectors of length one
- Use `str()` to see the structure of an object
- All more complex objects, like *S3*, *S4* or *Reference classes* are build from this structures
- Additionally R uses *attributes* to store metadata about an object

11.2 Data types

The usual types for an variable can be logical, numeric, double, character or complex (e.g. matrix, data.frame). It is quite handy to know your data types as then you know which operations/functions are available and meaningful to use and it saves you time when debugging

typeof	mode	storage.mode
logical	logical	logical
integer	numeric	integer
double	numeric	double
complex	complex	complex
character	character	character

In practice no distinction is made between doubles and integers. They are just "numerics".
Numeric are by default doubles:

```
typeof(1)
```

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

```
typeof(1L)
```

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

11.3 Assignments

Assignments in R are preferably of this form:

```
objectName <- value
```

It is also possible to assign with the *equal* sign.

```
objectName = value
```

```
x <- 5  # Both methods have the same outcome
y = 5
```

```
x
```

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

```
y
```

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

However, the “equal” sign is used for argument passing to functions. Thus if nesting, the equal sign will be interpreted as a argument assignment and might throw an error:

```
## We want to measure the running time of the inner product of a large vector and
## assign the outcome of the function to a variable simultaneously
system.time(a <- t(1:1e+06) %*% (1:1e+06))
```

```
##    user  system elapsed
##   0.060   0.036   0.096
```

```
system.time(a = t(1:1e+06) %*% (1:1e+06))  ## This would cause an error
```

```
## Error in system.time(a = t(1:1e+06) %*% (1:1e+06)): unused argument (a =
t(1:1e+06) %*% (1:1e+06))
```

Right Alt + - gives a quick shortcut to add **<-** ;)

11.4 Vectors

Vectors are 1-dimensional data structures which can contain one or more variables, regardless of their type.

Usually created with `c()` (from *concatenation*):

```
c(1, 5, 8, 10)
```

```
## [1]  1  5  8 10
```

```
str(c(1, 5, 8, 10))
```

```
##  num [1:4] 1 5 8 10
```

```
length(c(1, 5, 8, 10))
```

```
## [1] 4
```

```
c("a", "B", "cc")
```

```
## [1] "a"  "B"  "cc"
```

```
c(TRUE, FALSE, c(TRUE, TRUE))
```

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

```
c(1, "B", FALSE)
```

```
## [1] "1"     "B"     "FALSE"
```

There are multiple ways to create a numeric sequence depending on the desired result.

Usually for an integer sequence *from:to* is enough. For different results we may need to use `seq()` function utilizing its arguments to increase by a *step* of our choice or to split the range depending on the desired length of the output.

```
1:10
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

```r
seq(from = 1, to = 10, by = 0.3)
```

```
##  [1]  1.0  1.3  1.6  1.9  2.2  2.5  2.8  3.1  3.4  3.7  4.0  4.3  4.6  4.9  5.2
## [16]  5.5  5.8  6.1  6.4  6.7  7.0  7.3  7.6  7.9  8.2  8.5  8.8  9.1  9.4  9.7
## [31] 10.0
```

```r
seq(from = 1, to = 10, length.out = 12)
```

```
##  [1]  1.000000  1.818182  2.636364  3.454545  4.272727  5.090909  5.909091
##  [8]  6.727273  7.545455  8.363636  9.181818 10.000000
```

Keep in mind that sometimes different approaches, give different data types as results so when weird things happen, always check the documentation (devil is in the details)!

For non numeric values e.g. logical, character, `rep()` function comes also in handy.

The *replicate* function `rep()` replicates a vector a certain number of *times* and concatenates them:

```r
rep(c(TRUE, FALSE), times = 5)
```

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

To replicate *each* entry of the input vector at the time:

```r
rep(c(TRUE, FALSE), each = 5)
```

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

Vector elements are accessed via the `[` operator:

```r
## Create an A,B,C,D,E vector
x <- LETTERS[1:5]
x
```

```
## [1] "A" "B" "C" "D" "E"
```

```r
## access the third entry
x[3]
```

```
## [1] "C"
```

```r
## modify the third entry
x[3] <- "Z"
x
```

```
## [1] "A" "B" "Z" "D" "E"
```

Elements in vectors can have names. Using names instead of index to access entries in a vector make code more robust to re-ordering, sub-setting, and changes in data input.
Names can be created at initialization or set afterwards with `names()`.

```r
x <- c(a = 1, b = 2, c = 3)
x
```

```
## a b c
## 1 2 3
```

```r
names(x) <- c("A", "B", "C")
x
```

```
## A B C
## 1 2 3
```

Names don't have to be unique, but should preferably be, as sub-setting by names will only return the first match

```r
x <- c(a = 1, a = 2, b = 3)
x["a"]
```

```
## a
## 1
```

Not all elements need to have names.

```r
c(a = 1, 2, 3)
```

```
## a
## 1 2 3
```

11.5 Working with missing values

Missing values are specified with `NA` (Not Available). They are placeholders for the specific type and as such are something like an unspecified value. If not taken care of, it can break computation, e.g. What is the sum of [1, 3, 5, x] if you do not know x?

```r
## Define a vector containing a missing value (NA)
v <- c(1, 3, 5, NA)
```

Now we try to compute its mean:

```r
mean(v)
```

```
## [1] NA
```

For many functions including `mean()`, a `na.rm` parameter allows ignoring missing values:

```r
mean(v, na.rm = TRUE)
```

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

11.6 Factors

```r
x <- factor(c("red", "yellow", "red", "green", "green"))
x
```

```
## [1] red    yellow red    green  green
## Levels: green red yellow
```

- A factor is a is used to store categorical data. The distinct categories are called ‘levels’.
- They belong to a special `class()`, *factor*, which makes them behave differently from regular integer vectors.
- Factors cannot be altered in the same way as vectors. Only their levels can be altered.
- Factors are built on top of integer vectors: the values are integer indexes in the dictionary of levels.
- They occupy less space in memory than characters, since they are stored as integers.

Level	Integer
green	1
red	2
yellow	3

- Factors are typically constructed with `factor()`. By default the levels are the unique values, sorted by alphanumerical order.

```r
x <- factor(c("red", "yellow", "red", "green", "green"))
x
```

```
## [1] red    yellow red    green  green
## Levels: green red yellow
```

`levels()` gives the levels in ascending order of the underlying encoding integers.

```r
levels(x)
```

```
## [1] "green"  "red"    "yellow"
```

The order of the levels can be forced:

```r
x <- factor(c("red", "yellow", "red", "green", "green"), levels = c("green", "yellow",
    "red"))
x
```

```
## [1] red    yellow red    green  green
## Levels: green yellow red
```

```r
levels(x)
```

```
## [1] "green"  "yellow" "red"
```

The order of the levels is then used for all function requiring comparisons, e.g. sorting

```r
sort(x)
```

```
## [1] green  green  yellow red    red
## Levels: green yellow red
```

Only level values can be used:

```r
x <- factor(c("red", "yellow", "red", "green", "green"))
x[2] <- "blue"
x
```

```
## [1] red   <NA>  red   green green
## Levels: green red yellow
```

Be aware that R does not prevent you from combining factors:

Do not try to combine factors, especially if levels are not the same!

```r
c(x, factor("blue"))
```

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

11.7 Math

R supports various mathematical and logical operations between scalars, vectors etc.

- Basic mathematical operations: +, *, -, /
- Additional mathematical operations:
  - **%*** Matrix multiplication, for vectors: inner product (dot product)
  - **%%** Modulo
  - **%/%** Integer division
  - **%o%** Outer product
  - **ˆ** Exponentiation
- Boolean operations:
  - **&** element-wise AND
  - **&&** AND left to right until the result is determined

  - | element-wise OR
  - || OR left to right until the result is determined
  - **!** NOT
  - **xor(x,y)** element-wise XOR (exclusive OR)

11.8 Binary comparison

- Element-wise Binary comparison return a vector of same length as the input
  - **==** element-wise equality
  - **!=** element-wise inequality
  - **<**, **<=** element-wise smaller (or equal)
  - **>**, **>=** element-wise greater (or equal)

```
1:5 == 1:5
```

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

- Binary comparison with a single Boolean statement as output
  - `identical(x, y)` exact equality
  - `all(x)` are all TRUE?

```
identical(1:5, 1:5)
```

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

11.9 Matrices

2-dimensional structures that can be created by `matrix()`. By default 2D structures are populated column-wise.

```
mat <- matrix(data = 1:12, nrow = 2, ncol = 6, byrow = FALSE)
mat
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6]
## [1,]    1    3    5    7    9   11
## [2,]    2    4    6    8   10   12
```

```
mat <- matrix(data = 1:12, nrow = 2, ncol = 6, byrow = TRUE)
mat
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6]
## [1,]    1    2    3    4    5    6
## [2,]    7    8    9   10   11   12
```

```
# work the same as matrix(data = 1:12, nrow = 2, ncol = 6, byrow = FALSE)
mat <- matrix(1:12, 2, 6)
mat
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6]
## [1,]    1    3    5    7    9   11
## [2,]    2    4    6    8   10   12
```

```
dim(mat)
```

```
## [1] 2 6
```

Vector attributes generalizes differently for matrices as dimensions change:

`length()` becomes `nrow()` and `ncol()`

```
nrow(mat)
```

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

```
ncol(mat)
```

```
## [1] 6
```

`names()` becomes `rownames()` and `colnames()`

```
colnames(mat) <- c("A", "B", "C", "D", "E", "F")
rownames(mat) <- c("a", "b")
mat
```

```
##   A B C D  E  F
## a 1 3 5 7  9 11
## b 2 4 6 8 10 12
```

`c()` becomes `cbind()` (column-bind)

```
mat <- matrix(c(1, 2, 4, 5), 2, 2)
cmat <- matrix(7:8, 2, 1)
mat <- cbind(mat, cmat)
mat
```

```
##      [,1] [,2] [,3]
## [1,]    1    4    7
## [2,]    2    5    8
```

and `rbind()` (row-bind)

```
rmat <- matrix(c(3, 6, 9), 1, 3)
mat <- rbind(mat, rmat)
mat
```

```
##      [,1] [,2] [,3]
## [1,]    1    4    7
## [2,]    2    5    8
## [3,]    3    6    9
```

Operations between a vector and a matrix is done column-wise:

```
mat <- matrix(1, 3, 2)
```

```
vec <- 1:3
vec
```

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

```
mat * vec
```

`Can you guess what will happen?`

Operations between a vector and a matrix is done column-wise:

```
mat <- matrix(1:9, 3, 3)
```

```
vec <- 1:3
vec
```

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

```
mat * vec
```

```
##      [,1] [,2] [,3]
## [1,]    1    4    7
## [2,]    4   10   16
## [3,]    9   18   27
```

- This behavior comes from the fact that all objects in R are basically vectors, which are just "folded" column-wise
- Elements of the shorter vector are constantly reused.

```
as.vector(mat) * vec
```

```
## [1]  1  4  9  4 10 18  7 16 27
```

```
mat * 1:2
```

```
##      [,1] [,2] [,3]
## [1,]    1    8    7
## [2,]    4    5   16
## [3,]    3   12    9
```

- Elements of the shorter vector are constantly reused. If the shorter vector is a multiple of the longer vector, the computation succeeds. Otherwise it will run until the end and throw a warning.

```
mat * 1:4
```

```
##      [,1] [,2] [,3]
## [1,]    1   16   21
## [2,]    4    5   32
## [3,]    9   12    9
```

11.10 Lists

Lists are heterogeneous vectors, that is the elements can be of *any type*, including lists (recursive). Construct lists by using `list()`:

```
x <- list(c(1, 2, 3), "some text", list(c(TRUE, FALSE)))
x
```

```
## [[1]]
## [1] 1 2 3
##
## [[2]]
## [1] "some text"
##
## [[3]]
## [[3]][[1]]
## [1]  TRUE FALSE
```

`typeof` list is *list*:

```
typeof(x)
```

```
## [1] "list"
```

List elements are accessed via the `[[` operator:

```
## Create an A,B,C list
x <- as.list(LETTERS[1:3])
x
```

```
## [[1]]
## [1] "A"
##
## [[2]]
## [1] "B"
##
```

```
## [[3]]
## [1] "C"
```

```r
## access the third element with `[[`
x[[3]]
```

```
## [1] "C"
```

```r
## modify the third element
x[[3]] <- "Z"
x[[3]]
```

```
## [1] "Z"
```

```r
## accessed with only a single `[` we get a list of the given index
x[3]
```

```
## [[1]]
## [1] "Z"
```

11.11 Data Frames

- *Data frames* are the most common way to store data
- It's a list of equal-length vectors
- Because of this 2-dimensional structure it shares properties of both a *list* and a *matrix*
- So a data frame has `rownames()` and `colnames()` equivalent to `names()`
- It has also `nrow()` and `ncol()` equivalent to `length()`
- `length()` actually returns the number of *list elements*, which is the column dimension
- Data frames are created with `data.frame()` and named vectors (also lists) or matrices as input

```r
df <- data.frame(number = 1:4, letter = letters[1:4])
df
```

```
##   number letter
## 1      1      a
## 2      2      b
## 3      3      c
## 4      4      d
```

Matrices and lists can be without names, but data frames not. If an unnamed vector or matrix is provided, names are set by default. The functions `colnames` and `rownames` can be used to view and set the names.

```r
rownames(df)
```

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

```r
rownames(df) <- paste0("row_", nrow(df):1)
rownames(df)
```

```
## [1] "row_4" "row_3" "row_2" "row_1"
```

- Merging works with `cbind()` and `rbind()` like in matrices and with `c()` like in lists.
- If combining column-wise, the number of rows must match

```r
df <- data.frame(number = 1:2, letter = letters[1:2])
cbind(df, data.frame(cont = 5:6))
```

```
##   number letter cont
## 1      1      a    5
## 2      2      b    6
```

- If combining row-wise, **the column names have to be identical**.

```r
rbind(df, data.frame(n = 3, l = letters[3]))  # This will cause and Error
```

```
## Error in match.names(clabs, names(xi)): names do not match previous names
```

```r
rbind(df, data.frame(number = 3, letter = letters[3]))
```

```
##   number letter
## 1      1      a
## 2      2      b
## 3      3      c
```

11.12 Sub-setting

- There are three accessing methods
  - **[** For vectors
  - **[[** For list-based structures
  - **$** Similar to the prior
- Six ways of sub-setting
  - By positive integers
  - By negative integers
  - By logical vectors
  - Empty sub-setting
  - By Zero
  - Character vector

11.12.1 Atomic vectors

```r
x <- LETTERS[1:5]
x
```

```
## [1] "A" "B" "C" "D" "E"
```

```r
## By positive integers
x[c(1, 3)]
```

```
## [1] "A" "C"
```

```r
## By negative integers
x[-c(1, 3)]
```

```
## [1] "B" "D" "E"
```

```r
## By logical vectors
x[c(TRUE, TRUE, TRUE, FALSE, FALSE)]
```

```
## [1] "A" "B" "C"
```

```r
## By logical vectors
x[x <= "C"]
```

```
## [1] "A" "B" "C"
```

`NA`s in the input vector will always produce an `NA` in the output:

```r
x[c(1, 2, NA)]
```

```
## [1] "A" "B" NA
```

We can also sub-setting by character vectors, if names are present:

```
names(x) <- letters[1:5]
x[c("a", "b")]
```

```
##   a   b
## "A" "B"
```

11.12.2 Lists

Lists behave just like atomic vectors. The output is a list

```
myList <- list(char = c("a", "b"), int = 1:5, logic = TRUE)
## By positive integers
myList[[1]]
```

```
## [1] "a" "b"
```

```
myList[1]
```

```
## $char
## [1] "a" "b"
```

```
## Character vector
myList["int"]
```

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

```
## By logical vectors
myList[c(FALSE, TRUE, FALSE)]
```

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

```
## By negative integers
myList[-c(1, 3)]
```

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

11.12.3 Matrices

```
mat <- matrix(1:9, 3, 3)
colnames(mat) <- LETTERS[1:3]
mat
```

```
##      A B C
## [1,] 1 4 7
## [2,] 2 5 8
## [3,] 3 6 9
```

Possible to mix different sub-setting styles

```
mat[c(2, 3), c(1, 2)]
```

```
##      A B
## [1,] 2 5
## [2,] 3 6
```

```r
mat[-1, -3]
```

```
##      A B
## [1,] 2 5
## [2,] 3 6
```

```r
mat[c(2, 3), c("A", "B")]
```

```
##      A B
## [1,] 2 5
## [2,] 3 6
```

```r
mat[c(FALSE, TRUE, TRUE), c(TRUE, TRUE, FALSE)]
```

```
##      A B
## [1,] 2 5
## [2,] 3 6
```

```r
mat[-1, 1:2]
```

```
##      A B
## [1,] 2 5
## [2,] 3 6
```

```r
mat[c(FALSE, TRUE, TRUE), c("A", "B")]
```

```
##      A B
## [1,] 2 5
## [2,] 3 6
```

11.12.4 Data Frames

Data frames share properties of lists and matrices. sub-setting by a single vector will behave as in lists. Note the number of elements in a list is the number of columns in the data frame. sub-setting by two vectors will behave as in matrices.

```r
df <- data.frame(matrix(1:6, 2, 2))
names(df) <- letters[1:2]
df[1:2]
```

```
##   a b
## 1 1 3
## 2 2 4
```

```r
df[-1, c("a", "b")]
```

```
##   a b
## 2 2 4
```

11.12.5 By accessors:

The `[[` and `$` operators behave more like accessors to objects. They return only one object and as such can take generally only one input, so they are useful for lists and data frames. The `$` is a shortcut for `x[[name, exact = FALSE]]`

```r
names(df) <- c("Ak", "B")
df
```

```
##   Ak B
## 1  1 3
## 2  2 4
```

```r
df[[2]]
```

```
## [1] 3 4
```

```r
df[["B"]]
```

```
## [1] 3 4
```

What will the following return?

```r
df$A
```

What will the following return?

```r
df$A
```

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

```r
df[["A", exact = TRUE]]
```

```
## NULL
```

11.12.6 Sub-setting and assignment

All sub-setting operators can be combined with assignments to modify a subset of values.

```r
x <- c(a = 1, b = 2, c = 3)
x[2] <- 10
x
```

```
##  a  b  c
##  1 10  3
```

```r
df <- data.frame(a = 1:2, b = 3:4)
df
```

```
##   a b
## 1 1 3
## 2 2 4
```

```r
df$b <- c("first", "second")
df
```

```
##   a      b
## 1 1  first
## 2 2 second
```

11.13 Functions

Function are a nice way to break down a big problem (bake a pizza) to smaller, manageable sub-problems (make the dough, make the sauce, add toppings, bake the pizza). They offer code re-usage (save time for important things, like wine drinking), easier testing of functionality and cleaner code!

The syntax to write your own functions is the following:

```r
myFunction <- function(args1, args2 = default, ...) {
    body
    return(result)
}
```

- If the `return` statement is missing, the result of the last line will be returned
- The parameters can be assigned default values
- If the default value is a vector, the first value will be used by default
- The *three dots* (`...`) in the function arguments are called **ellipsis** and are used as a placeholder for any arguments passed to a function call inside. This is useful when you don't want to push all parameters from embodied function to your functions or allow for generic functions without pre-specifying argument names.

```
myFunction <- function(args1, args2 = c("option1", "option2"), ...) {
    result <- doSomething(args1, ...)
    return(result)
}
myFunction(args1, args2, param1)
```

11.14 Conditional Statements

A typical way to control the flow of the execution of your code is to have conditional statements.

The typical setup is:

```
if (statement_to_test) {
    print("Statement is true")
} else {
    print("Statement is false")
}
```

Long version:

```
if (1 < 2) {
    print("TRUE")
} else {
    print("FALSE")
}
```

```
## [1] "TRUE"
```

Short version if the To-Do step fits into one line:

```
{
    if (1 > 2)
        print("TRUE") else print("FALSE")
}
```

```
## [1] "FALSE"
```

The curly brackets around the if-else statement are only needed in a script to tell R to which *if* the *else* belongs to.

You can also have multiple statement tests. If more than one statements can be satisfied the first that got evaluated will be executed:

```
x <- 1
if (x > 0) {
    print("X is 0, or maybe not?")
} else if (x == 1) {
    print("X is 1")
} else {
    print("X is something else")
}
```

```
## [1] "X is 0, or maybe not?"
```

You can also chain *if-else* statements:

```
x <- 5
if (x > 0) {
    if (x > 3) {
        if (x == 5) {
            cat("X is 5")
        } else {
            cat("X is greater than 3")
        }
    } else {
        cat("X positive and less than 3")
    }
} else {
    cat("X less than 0")
}
```

R also supports switch statements.

11.15 For loops

R supports looping through the traditional *for* and *while* statements and with the use of the *apply family* functions.

Syntax:

```
for(index in sequence) {
  do something
}
```

Again, if the To-Do step is a one-liner, curly brackets can be dropped

```
a <- NULL
for (i in 1:10) a <- c(a, sum(1:i))
a
```

```
##  [1]  1  3  6 10 15 21 28 36 45 55
```

A sequence can be any vector, including lists

```
myList <- list(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
a <- integer(10)
for (i in myList) a[i] <- sum(1:i)
a
```

```
##  [1]  1  3  6 10 15 21 28 36 45 55
```

However, usually looping using *for/while* statements is more time consuming compared to the the functions of the *apply family*.

11.16 The Apply family

- For loops condition each following step on the previous step. That means that after each step the environment is updated.
- This leads to long running times (for the already slow looping mechanism of R).
- In case the single steps are independent of each other, a particular function can be applied to the elements of the steps.

```
myList <- list(1:10, 11:20, 5:14)
sapply(myList, sum)
```

```
## [1]  55 155  95
```

- Also lambda functions can be applied

```
sapply(myList, function(y) {
    sum(y)/length(y)
})
```

```
## [1]  5.5 15.5  9.5
```

11.17 R Markdown

- This is an R Markdown presentation. Markdown is a simple formatting syntax for authoring HTML, PDF, and MS Word documents. For more details on using R Markdown see http://rmarkdown.rstudio.com.
- Simply go to File –> New File –> R Markdown
- Select PDF and you get a template.
- You most likely won't need more commands than in on the first page of this cheat sheet.
- When you click the **Knit** button a document will be generated that includes both content as well as the output of any embedded R code chunks within the document.

11.18 Useful references

- These slides are mostly based on Hadley Wickham's book Advanced R
- **Everything else** from Hadley Wickham
- In-depth documentations:
  - Introduction to R
  - R language definition
  - R Internals
- Last but not least:
  - Stackoverflow

Chapter 12

Appendix 3: Additonal plotting tools

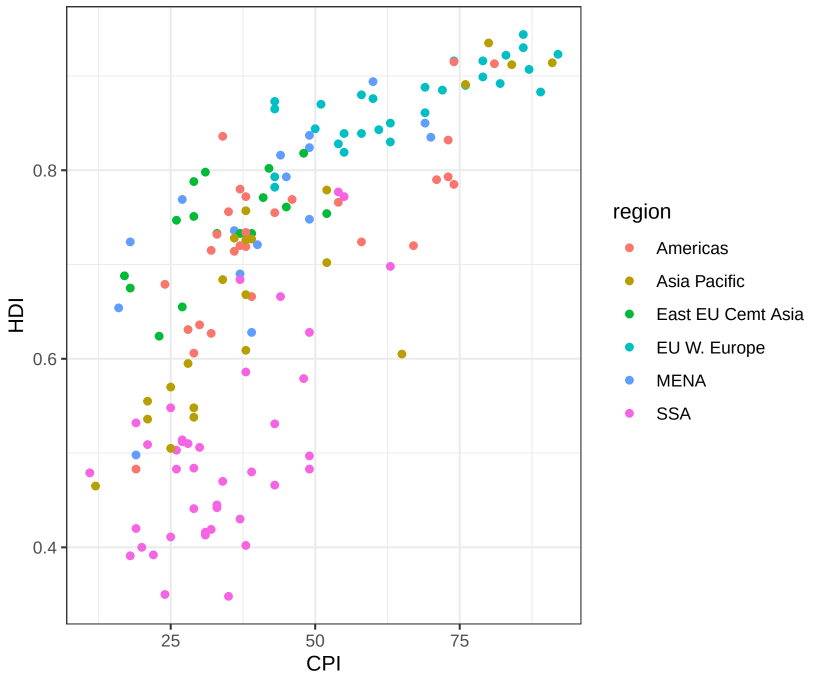
12.1 Plotting themes

Themes control non-data parts of your plots: * Complete theme * Axis title * Axis text * Plot title * Legends

They control the appearance of you plots: size, color, position…

But nothing to do with how the data is represented.

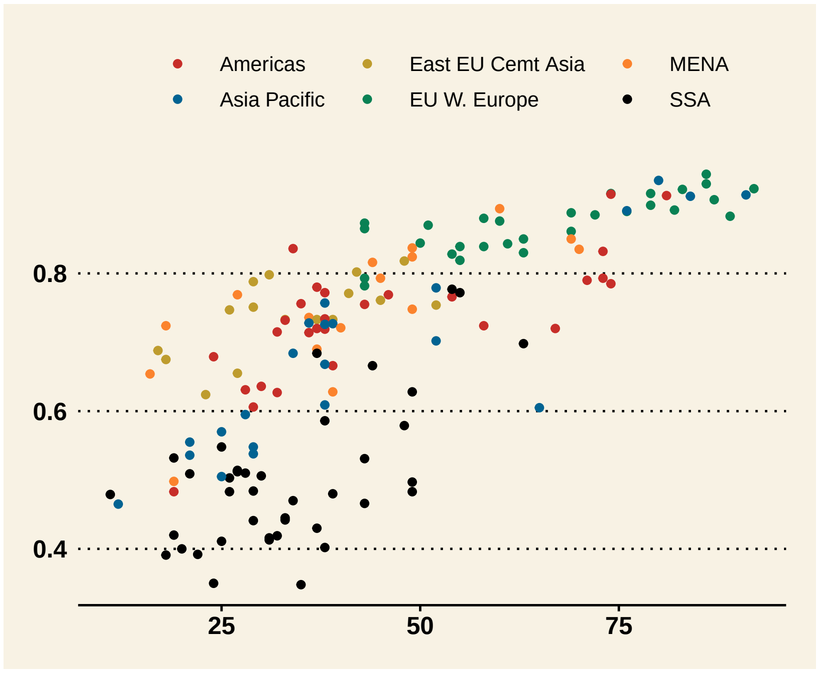
```
ggplot(ind, aes(CPI, HDI, color = region)) + geom_point() + theme_bw()
```



Other complete themes: `theme_classic`, `theme_minimal`, `theme_light`, `theme_dark`.

More themes with `ggthemes` package (from github)

```
library(ggthemes)
ggplot(ind, aes(CPI, HDI, color = region)) + geom_point() + theme_wsj() + scale_colour_wsj("colors6",
    "")
```



wsj: Wall Street Journal

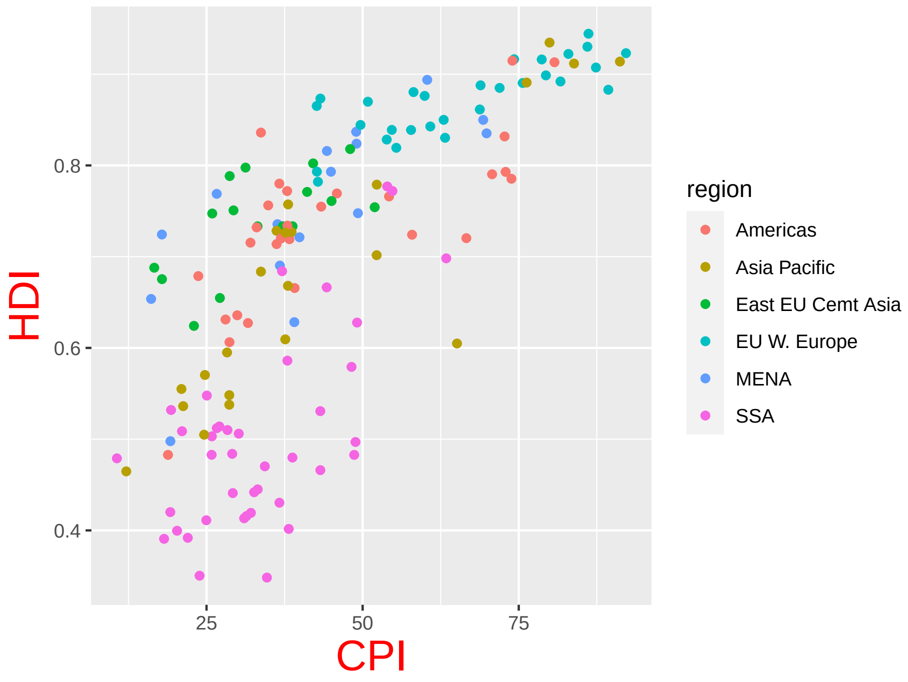
12.2 Axis titles

`axis.title`

`axis.title.x`

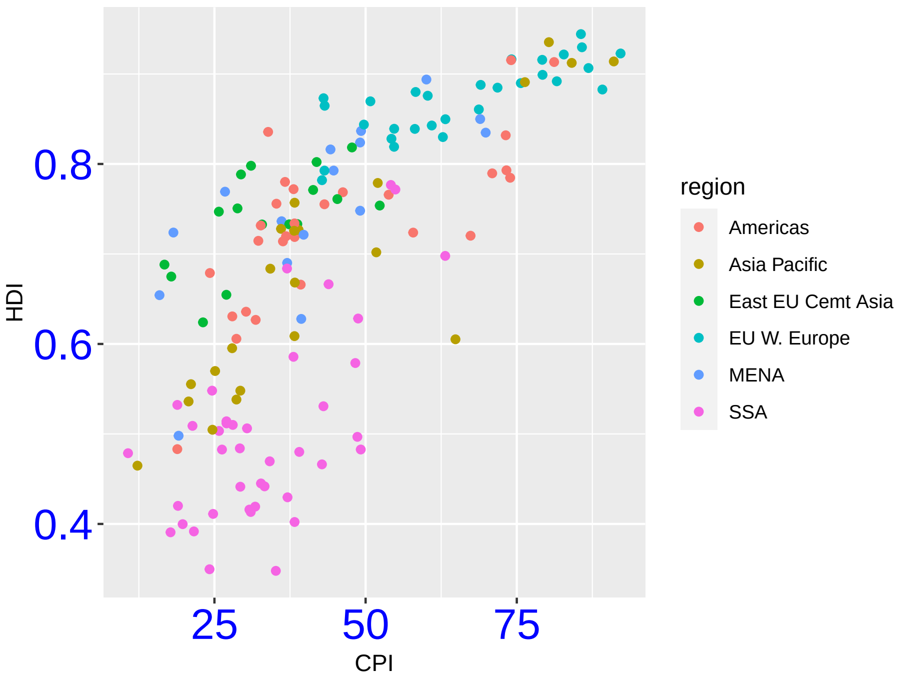
`axis.title.y`

```
ggplot(ind, aes(CPI, HDI, color = region)) + geom_jitter() + theme(axis.title = element_text(size = 20,
    color = "red"))
```



or

```
ggplot(ind, aes(CPI, HDI, color = region)) + geom_jitter() + theme(axis.text = element_text(size = 20,
    color = "blue"))
```



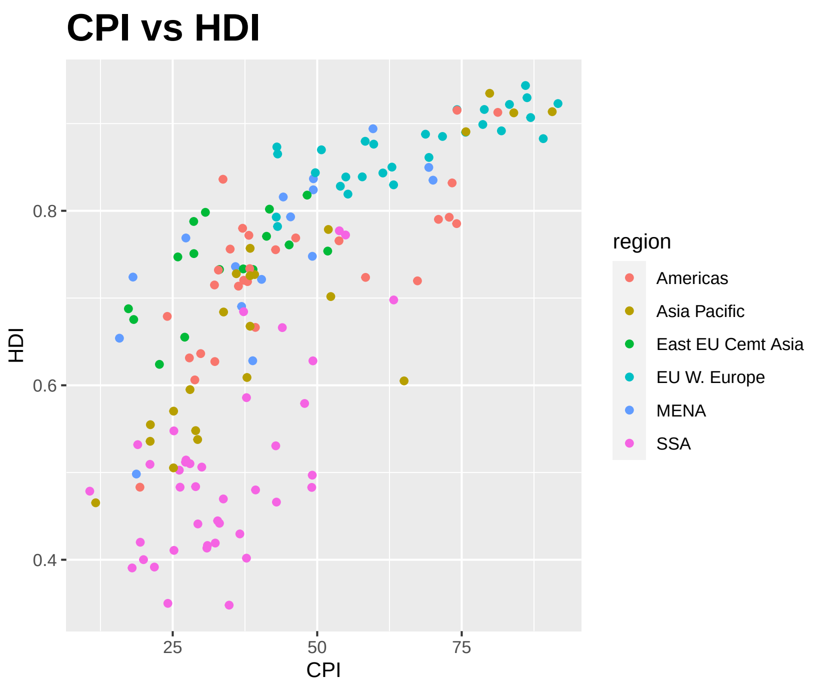
12.3 Summary axis elements

(don't have to know them by heart) The axis elements control the apperance of the axes:

Element	Setter	Description
axis.line	`element_line()`	line parallel to axis (hidden in default themes)
axis.text	`element_text()`	tick labels
axis.text.x	`element_text()`	x-axis tick labels
axis.text.y	`element_text()`	y-axis tick labels
axis.title	`element_text()`	axis titles
axis.title.x	`element_text()`	x-axis title
axis.title.y	`element_text()`	y-axis title
axis.ticks	`element_line()`	axis tick marks
axis.ticks.length	`unit()`	length of tick marks

12.3.1 Plot title themes

```
ggplot(ind, aes(CPI, HDI, color = region)) + geom_jitter() + ggtitle("CPI vs HDI") +
    theme(plot.title = element_text(size = 20, face = "bold"))
```

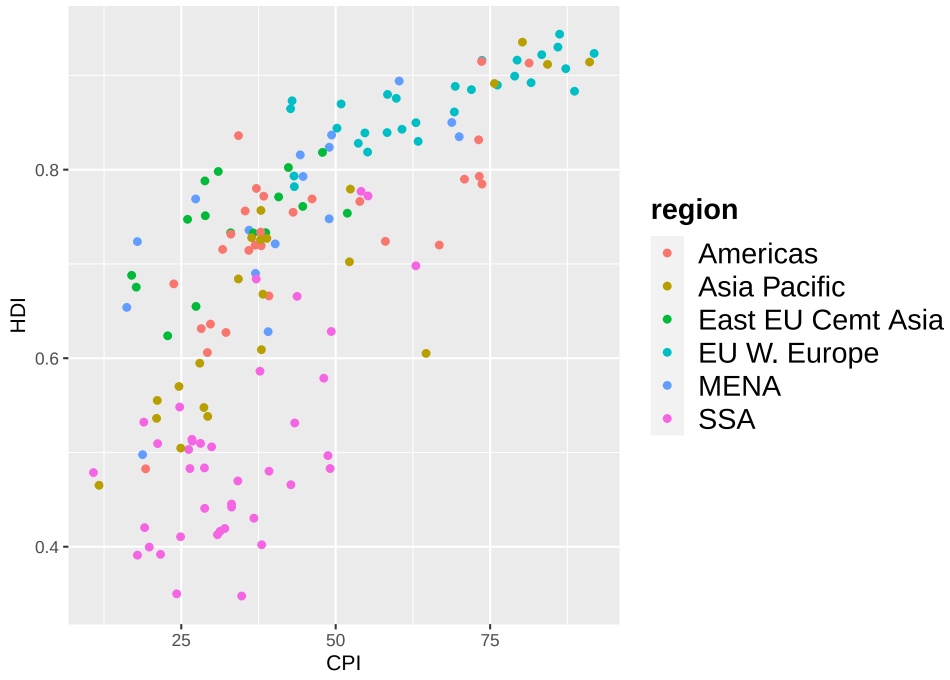


Some elements affect the plot as a whole:

Element	Setter	Description
plot.background	`element_rect()`	plot background
plot.title	`element_text()`	plot title
plot.margin	`margin()`	margins around plot

12.3.2 Legend themes (don't have to know them by heart)

```r
base <- ggplot(ind, aes(CPI, HDI, color = region)) + geom_jitter()
base + theme(legend.text = element_text(size = 15), legend.title = element_text(size = 15,
    face = "bold"))
```



The legend elements control the apperance of all legends. You can also modify the appearance of individual legends by modifying the same elements in `guide_legend()` or `guide_colourbar()`.

Element	Setter	Description
legend.background	`element_rect()`	legend background
legend.key	`element_rect()`	background of legend keys
legend.key.size	`unit()`	legend key size
legend.key.height	`unit()`	legend key height
legend.key.width	`unit()`	legend key width
legend.margin	`unit()`	legend margin
legend.text	`element_text()`	legend labels
legend.text.align	0–1	legend label alignment (0 = right, 1 = left)
legend.title	`element_text()`	legend name
legend.title.align	0–1	legend name alignment (0 = right, 1 = left)