Data Science and Data Analytics

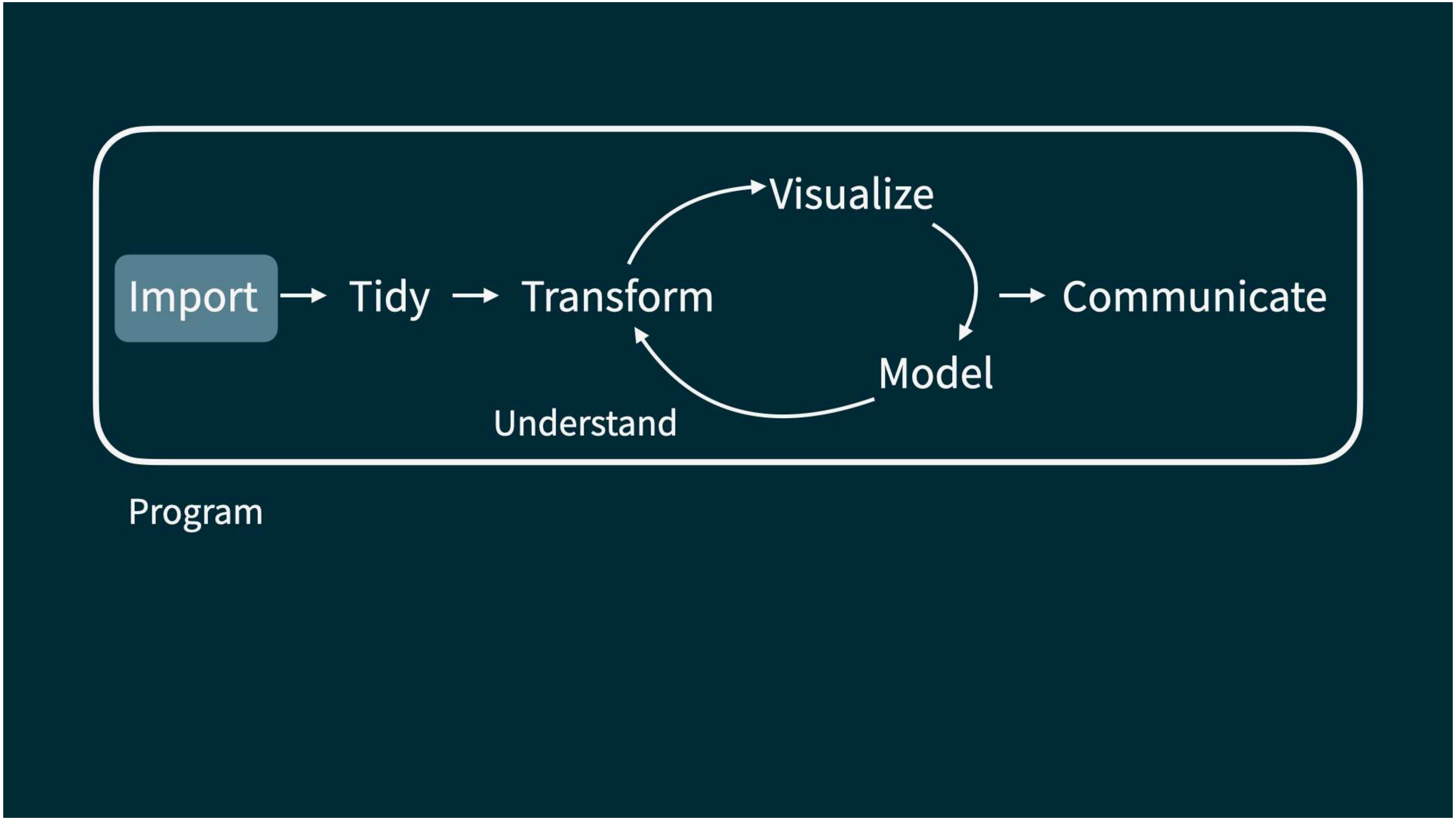
The Data Science Workflow I – Import, Tidy and Transform

Julian Amon, PhD
Charlotte Fresenius Privatuniversität

March 31, 2025

Import

The Data Science workflow – Import

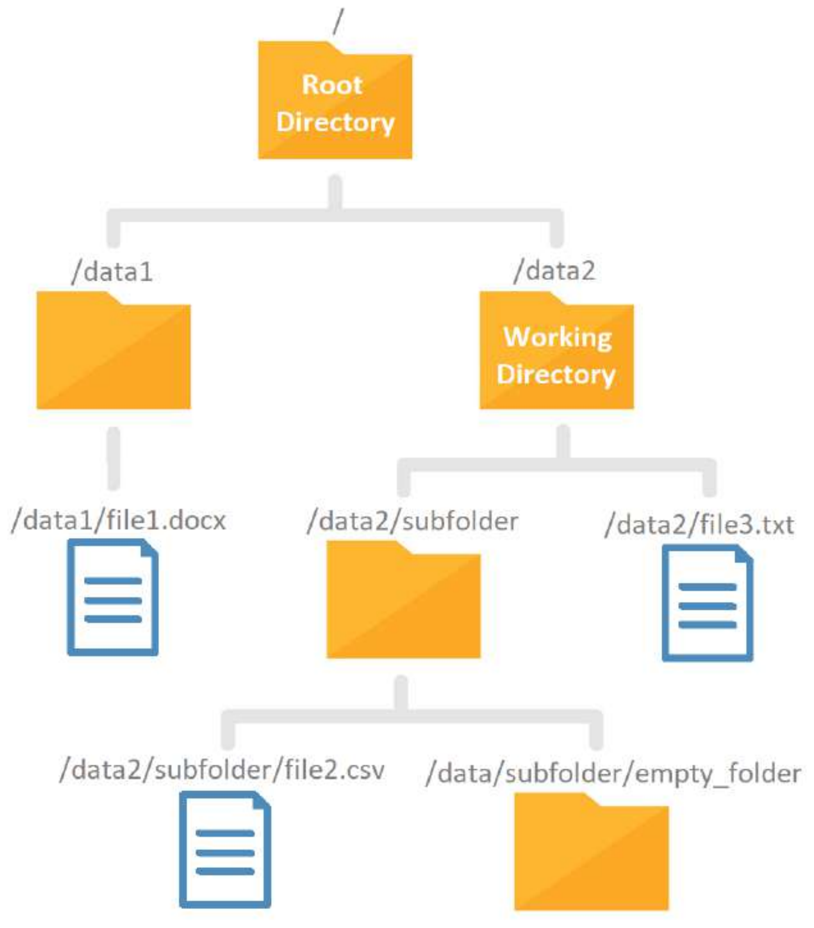


Importing data into R

- In order to get our data science workflow started, we need to be able to **import** data from different sources into R.
- While there is a huge number of different data formats, we will focus on how to import data stored in two of the most common types of formats, namely:
  - Text files (like `csv` or `tsv`)
  - Spreadsheets (Excel or Google Sheets)
- Before we dive into how to get the data stored in such files into R, we need to be able to find these files on our computer in the first place…
- For this, we first have to look into how R orients itself in the **file system** on our computer.

The file system

- A computer's file system consists of nested folders (*directories*). It can be visualized as tree structures, with directories branching out from the root.
- The **root directory** contains all other directories.
- The **working directory** is the current location in the filesystem.



Relative and full paths

- A **path** lists directory names leading to a file. Think of it like instructions on what folders to click on, and in what order, to find the file. We distinguish:
  - **Full paths**: Starts from the root directory, i.e. the very top of the file system hierarchy. An example would be:

```r
example_path <- system.file(package = "dslabs")
example_path
```

```
[1] "/home/julian/R/x86_64-pc-linux-gnu-library/4.4/dslabs"
```

  - **Relative paths**: Starts from the current working directory. Imagine the current working directory would be `/home/julian`, then the **relative** path to the folder above would simply be:

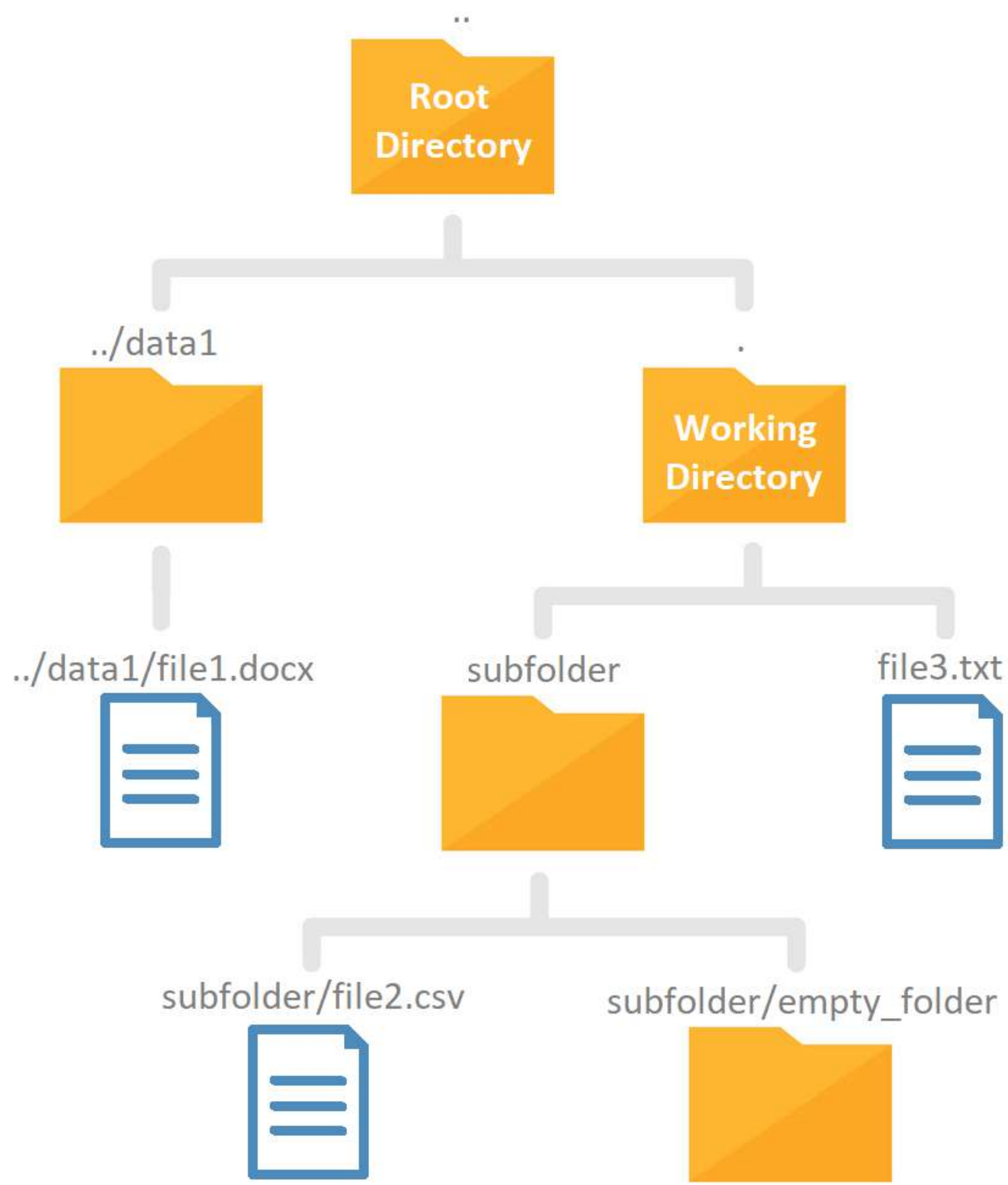
```
[1] "R/x86_64-pc-linux-gnu-library/4.4/dslabs"
```

- In R, we can use the `list.files` function to explore directories:

```r
list.files(example_path)
```

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

Relative and full paths



The working directory

- When referring to files in your R script, it is highly recommended that you use **relative paths**.
- **Reason**: paths are unique to your computer, so if someone else runs your code on their computer, it will not find files whose location is described by absolute paths.
- To determine the working directory of your current R session, you can type:

```r
getwd()
```

```
[1] "/home/julian/Dokumente/Projekte/Lehre/CFPU Data Science und Data Analytics/2025_SS"
```

- To change the working directory, use the function `setwd`:

```r
setwd("/path/to/your/directory")
```

In RStudio, you can alternatively also select the working directory via `Session > Set Working Directory`.

Generating path names

- Different operating systems have different conventions when specifying paths.
- For example, Linux and Mac use forward slashes `/`, while Windows uses backslashes `\` to separate directories.
- The R function `file.path` combines characters to form a complete path, automatically ensuring compatibility with the respective operating system.
- This function is useful because we often want to define paths using a variable.
- Consider the following example:

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

```
[1] "/home/julian/R/x86_64-pc-linux-gnu-library/4.4/dslabs/extdata/murders.csv"
```

Here the variable `dir` contains the full path for the `dslabs` package (needs to be installed!) and `extdata/murders.csv` is the relative path of a specific `csv` file in that folder.

Importing data from text files

- All of us know **text files**. They are easy to open, can be easily read by humans and are easily transferable.
- When text files are used to store **tabular data**, line breaks are used to separate rows and a predefined character (the so-called **delimiter**) is used to separate columns within a row. Which one is used can depend on the file format:
  - `.csv` (comma-separated values) typically uses comma (`,`) or semicolon (`;`).
  - `.tsv` (tab-separated values) typically uses tab (which can be a preset number of spaces or `\t`).
  - `.txt` ("text") can use any of the above or a simple space ( )
- How we read text files into R depends on the delimiter used. Therefore, we need to have a look at the file to determine the delimiter.

Importing data from text files

This is the first couple of lines of the `murders.csv` text file from the `dslabs` package we saw referenced before. It contains the number of gun murders in each US state in the year 2010 as well as each state's population. Clearly, it uses commas (`,`) as delimiter. Also note the use of a **header** in the first row.

murders.csv

```
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
```

Importing data from text files – `.csv`

- For **comma-delimited** `csv` files, R offers the function `read.csv` to run on the (full or relative) path of the file. By default, it assumes that **decimal points** are used and that a **header** giving column names is present:

```
args(read.csv)
```

```
function (file, header = TRUE, sep = ",", quote = "\"", dec = ".",
    fill = TRUE, comment.char = "", ...)
NULL
```

- For **semicolon-delimited** `csv` files, R offers the function `read.csv2` to run on the (full or relative) path of the file. By default, it assumes that **decimal commas** are used and that a **header** giving column names is present.

```
args(read.csv2)
```

```
function (file, header = TRUE, sep = ";", quote = "\"", dec = ",",
    fill = TRUE, comment.char = "", ...)
NULL
```

- Both of these functions return a `data.frame` containing the data from the file.

Importing data from text files – `.csv`

As `murders.csv` is comma-delimited, we use `read.csv` to read it into R:

```
dir <- system.file(package = "dslabs")
murders_df <- read.csv(file.path(dir, "extdata", "murders.csv"))
head(murders_df, 6) # shows the first 6 rows of the data frame
```

```
       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
```

Note that the categorical variables (`state`, `abb` and `region`) are imported as `character` vectors:

```
unlist(lapply(murders_df, typeof))
```

```
      state         abb      region  population       total 
"character" "character" "character"   "integer"   "integer" 
```

For data analysis purposes, we should probably turn these into factors. But for now, we are only interested in the successful import.

Importing data from text files – `.tsv`

- For **tab-delimited** `tsv` files, R offers the functions `read.delim` and `read.delim2`, again assuming the use of **decimal points** and **decimal commas**, respectively:

```r
args(read.delim)
```

```
function (file, header = TRUE, sep = "\t", quote = "\"", dec = ".",
    fill = TRUE, comment.char = "", ...)
NULL
```

```r
args(read.delim2)
```

```
function (file, header = TRUE, sep = "\t", quote = "\"", dec = ",",
    fill = TRUE, comment.char = "", ...)
NULL
```

- Otherwise, the use is identical to `read.csv`: the function requires the path to the file you want to import and returns a `data.frame` containing the (hopefully) correctly parsed data from the file.

Importing data from text files – `.tsv`

mtcars.tsv - Editor

Datei Bearbeiten Format Ansicht Hilfe

```
model	mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
Mazda RX4	21	6	160	110	3.9	2.62	16.46	0	1	4	4
Mazda RX4 Wag	21	6	160	110	3.9	2.875	17.02	0	1	4	4
Datsun 710	22.8	4	108	93	3.85	2.32	18.61	1	1	4	1
Hornet 4 Drive	21.4	6	258	110	3.08	3.215	19.44	1	0	3	1
Hornet Sportabout	18.7	8	360	175	3.15	3.44	17.02	0	0	3	2
Valiant	18.1	6	225	105	2.76	3.46	20.22	1	0	3	1
Duster 360	14.3	8	360	245	3.21	3.57	15.84	0	0	3	4
Merc 240D	24.4	4	146.7	62	3.69	3.19	20	1	0	4	2
Merc 230	22.8	4	140.8	95	3.92	3.15	22.9	1	0	4	2
Merc 280	19.2	6	167.6	123	3.92	3.44	18.3	1	0	4	4
Merc 280C	17.8	6	167.6	123	3.92	3.44	18.9	1	0	4	4
Merc 450SE	16.4	8	275.8	180	3.07	4.07	17.4	0	0	3	3
Merc 450SL	17.3	8	275.8	180	3.07	3.73	17.6	0	0	3	3
Merc 450SLC	15.2	8	275.8	180	3.07	3.78	18	0	0	3	3
Cadillac Fleetwood	10.4	8	472	205	2.93	5.25	17.98	0	0	3	4
Lincoln Continental	10.4	8	460	215	3	5.424	17.82	0	0	3	4
Chrysler Imperial	14.7	8	440	230	3.23	5.345	17.42	0	0	3	4
Fiat 128	32.4	4	78.7	66	4.08	2.2	19.47	1	1	4	1
Honda Civic	30.4	4	75.7	52	4.93	1.615	18.52	1	1	4	2
Toyota Corolla	33.9	4	71.1	65	4.22	1.835	19.9	1	1	4	1
Toyota Corona	21.5	4	120.1	97	3.7	2.465	20.01	1	0	3	1
Dodge Challenger	15.5	8	318	150	2.76	3.52	16.87	0	0	3	2
AMC Javelin	15.2	8	304	150	3.15	3.435	17.3	0	0	3	2
Camaro Z28	13.3	8	350	245	3.73	3.84	15.41	0	0	3	4
Pontiac Firebird	19.2	8	400	175	3.08	3.845	17.05	0	0	3	2
Fiat X1-9	27.3	4	79	66	4.08	1.935	18.9	1	1	4	1
Porsche 914-2	26	4	120.3	91	4.43	2.14	16.7	0	1	5	2
Lotus Europa	30.4	4	95.1	113	3.77	1.513	16.9	1	1	5	2
Ford Pantera L	15.8	8	351	264	4.22	3.17	14.5	0	1	5	4
Ferrari Dino	19.7	6	145	175	3.62	2.77	15.5	0	1	5	6
Maserati Bora	15	8	301	335	3.54	3.57	14.6	0	1	5	8
Volvo 142E	21.4	4	121	109	4.11	2.78	18.6	1	1	4	2
```

Zeile 1, Spalte 1 | 100% | Unix (LF) | UTF-8

→ `read.delim`

```
> read.delim("mtcars.tsv")
                 model  mpg cyl  disp  hp drat    wt  qsec vs am gear carb
1            Mazda RX4 21.0   6 160.0 110 3.90 2.620 16.46  0  1    4    4
2        Mazda RX4 Wag 21.0   6 160.0 110 3.90 2.875 17.02  0  1    4    4
3           Datsun 710 22.8   4 108.0  93 3.85 2.320 18.61  1  1    4    1
4       Hornet 4 Drive 21.4   6 258.0 110 3.08 3.215 19.44  1  0    3    1
5    Hornet Sportabout 18.7   8 360.0 175 3.15 3.440 17.02  0  0    3    2
6              Valiant 18.1   6 225.0 105 2.76 3.460 20.22  1  0    3    1
7           Duster 360 14.3   8 360.0 245 3.21 3.570 15.84  0  0    3    4
8            Merc 240D 24.4   4 146.7  62 3.69 3.190 20.00  1  0    4    2
9             Merc 230 22.8   4 140.8  95 3.92 3.150 22.90  1  0    4    2
10            Merc 280 19.2   6 167.6 123 3.92 3.440 18.30  1  0    4    4
11           Merc 280C 17.8   6 167.6 123 3.92 3.440 18.90  1  0    4    4
12          Merc 450SE 16.4   8 275.8 180 3.07 4.070 17.40  0  0    3    3
13          Merc 450SL 17.3   8 275.8 180 3.07 3.730 17.60  0  0    3    3
14         Merc 450SLC 15.2   8 275.8 180 3.07 3.780 18.00  0  0    3    3
15  Cadillac Fleetwood 10.4   8 472.0 205 2.93 5.250 17.98  0  0    3    4
16 Lincoln Continental 10.4   8 460.0 215 3.00 5.424 17.82  0  0    3    4
17   Chrysler Imperial 14.7   8 440.0 230 3.23 5.345 17.42  0  0    3    4
18            Fiat 128 32.4   4  78.7  66 4.08 2.200 19.47  1  1    4    1
19         Honda Civic 30.4   4  75.7  52 4.93 1.615 18.52  1  1    4    2
20      Toyota Corolla 33.9   4  71.1  65 4.22 1.835 19.90  1  1    4    1
21       Toyota Corona 21.5   4 120.1  97 3.70 2.465 20.01  1  0    3    1
22    Dodge Challenger 15.5   8 318.0 150 2.76 3.520 16.87  0  0    3    2
23         AMC Javelin 15.2   8 304.0 150 3.15 3.435 17.30  0  0    3    2
24          Camaro Z28 13.3   8 350.0 245 3.73 3.840 15.41  0  0    3    4
25    Pontiac Firebird 19.2   8 400.0 175 3.08 3.845 17.05  0  0    3    2
26           Fiat X1-9 27.3   4  79.0  66 4.08 1.935 18.90  1  1    4    1
27       Porsche 914-2 26.0   4 120.3  91 4.43 2.140 16.70  0  1    5    2
28        Lotus Europa 30.4   4  95.1 113 3.77 1.513 16.90  1  1    5    2
29      Ford Pantera L 15.8   8 351.0 264 4.22 3.170 14.50  0  1    5    4
30        Ferrari Dino 19.7   6 145.0 175 3.62 2.770 15.50  0  1    5    6
31       Maserati Bora 15.0   8 301.0 335 3.54 3.570 14.60  0  1    5    8
32          Volvo 142E 21.4   4 121.0 109 4.11 2.780 18.60  1  1    4    2
```

Importing data from text files – `.txt`

- In fact, all of the functions for importing data discussed so far are just interfaces to the R function `read.table`, which provides the most flexibility when importing data from text files:

```
args(read.table)
```

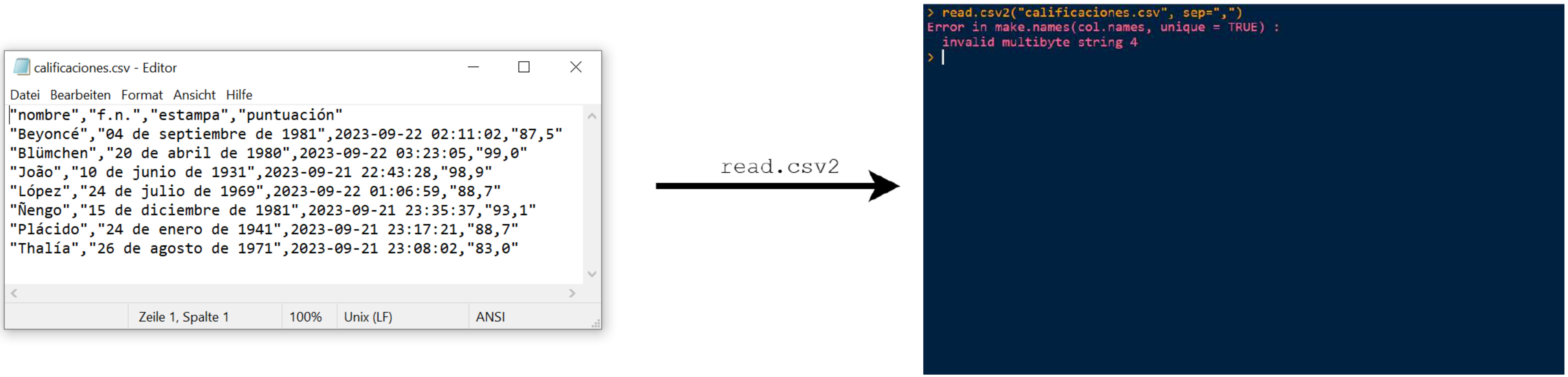
```
function (file, header = FALSE, sep = "", quote = "\"'", dec = ".",
    numerals = c("allow.loss", "warn.loss", "no.loss"), row.names,
    col.names, as.is = !stringsAsFactors, tryLogical = TRUE,
    na.strings = "NA", colClasses = NA, nrows = -1, skip = 0,
    check.names = TRUE, fill = !blank.lines.skip, strip.white = FALSE,
    blank.lines.skip = TRUE, comment.char = "#", allowEscapes = FALSE,
    flush = FALSE, stringsAsFactors = FALSE, fileEncoding = "",
    encoding = "unknown", text, skipNul = FALSE)
NULL
```

  (As always, to see the meaning of all of these arguments, see `?read.table`)

- This function is mostly used directly, when importing data from generic `.txt` files, where the format is often less strictly adhered to than in `csv` or `tsv` files.

Encoding

Now, we know how to import data into R. So we download some data set, read it into R using the correct function, but then this happens…



UNIVERSITY OF SUSTAINABILITY

Encoding

- Such issues occur because of an incorrectly identified **file encoding**.
- Encoding refers to how the computer stores character strings as binary 0s and 1s. Examples of encoding systems are:
  - **ASCII**: uses 7 bits to represent symbols, enough for all English keyboard characters, but not much more…
  - **Unicode** (especially UTF-8): the de-facto standard encoding of the internet, able to represent everything from the English alphabet to German Umlaute to Chinese characters and emojis.
- When reading a text file into R, its encoding needs to be known, otherwise the import either fails (see previous slide) or produces gibberish (e.g. German "Höhe" $\longrightarrow$ "HÃ¶he").

Encoding

- RStudio typically uses **UTF-8** as its default, which works in most cases. If it does not, you can use the `guess_encoding` function of the `readr` package to get insight into the encoding.

```
library(readr)
weird_filepath <- file.path("data", "calificaciones.csv")
as.data.frame(guess_encoding(weird_filepath))
```

```
    encoding confidence
1 ISO-8859-1       0.92
2 ISO-8859-2       0.72
3 ISO-8859-9       0.53
```

- The function deems the `ISO-8859-1` encoding to be the most likely encoding of the previous file. So we pass this value as the `fileEncoding` argument to `read.csv2`:

```
head(read.csv2(weird_filepath, sep = ",", fileEncoding = "ISO-8859-1"), 3)
```

```
    nombre                        f.n.             estampa puntuación
1  Beyoncé 04 de septiembre de 1981 2023-09-22 02:11:02       87.5
2 Blümchen       20 de abril de 1980 2023-09-22 03:23:05       99.0
3     João       10 de junio de 1931 2023-09-21 22:43:28       98.9
```

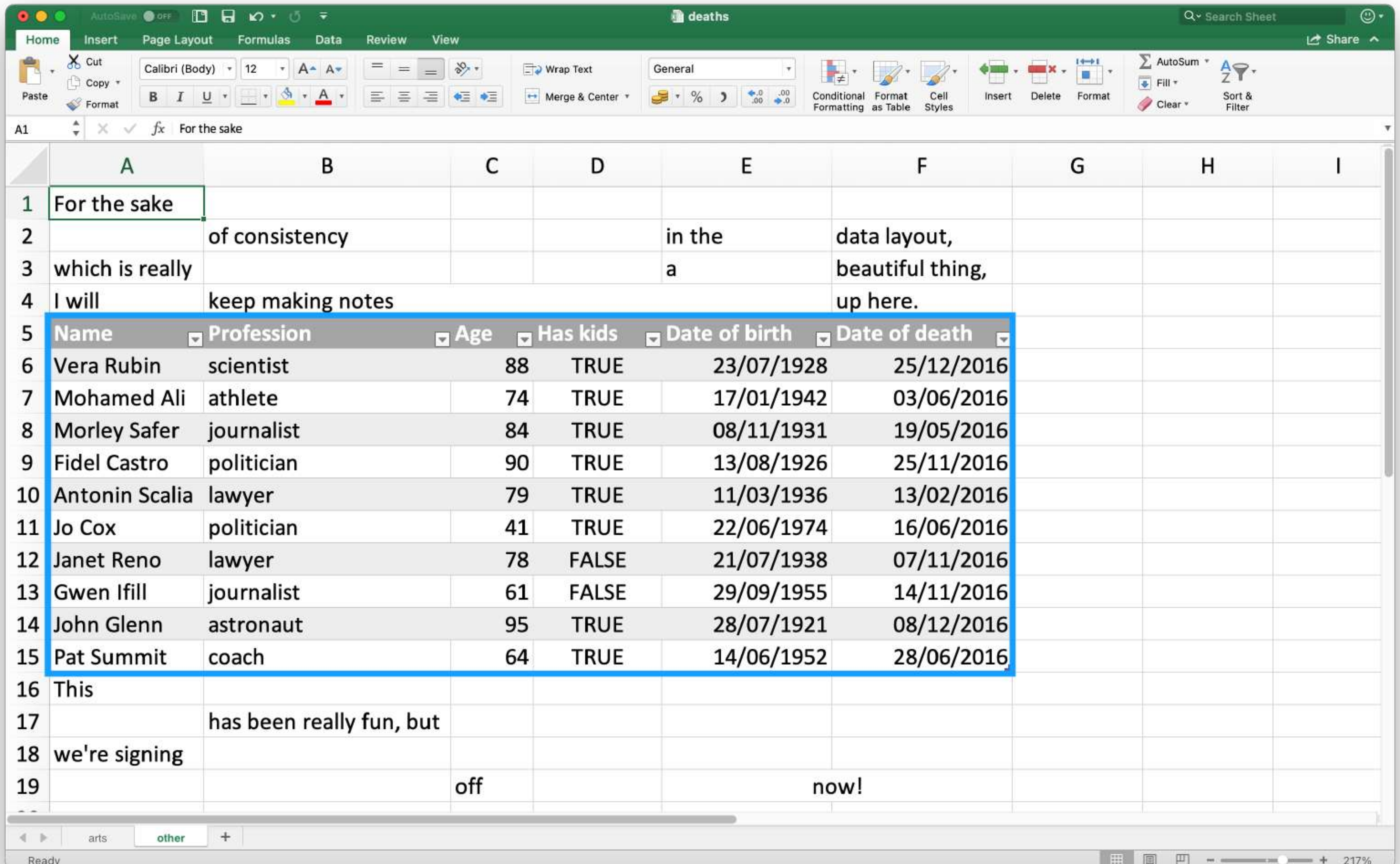
- This time, it worked! 😊

Importing data from spreadsheets

- Another common way of sharing tabular data is through the use of **spreadsheets**, like Excel or Google Sheets. We will see how to import data in both of these types of documents.
- With Excel, spreadsheets typically either have a `.xls` or `.xlsx` file suffix. Note that those are **binary** file formats, i.e. unlike text files, they are not human-readable when opened with a text editor.
- Base R does *not* have functionality to import data from Excel spreadsheets. However, the package `readxl` does. Its two main functions are:
  - `read_xls` to read Excel spreadsheets with `.xls` ending and
  - `read_xlsx` to read Excel spreadsheets with `.xlsx` ending.
- These functions allow to select only certain areas of certain sheets, transform data types and much more…

Importing data from spreadsheets

Consider the following simple example. Suppose we have the following `.xlsx`-spreadsheet of famous people that died in 2016:



	A	B	C	D	E	F
1	For the sake					
2		of consistency			in the	data layout,
3	which is really				a	beautiful thing,
4	I will	keep making notes				up here.
5	Name	Profession	Age	Has kids	Date of birth	Date of death
6	Vera Rubin	scientist	88	TRUE	23/07/1928	25/12/2016
7	Mohamed Ali	athlete	74	TRUE	17/01/1942	03/06/2016
8	Morley Safer	journalist	84	TRUE	08/11/1931	19/05/2016
9	Fidel Castro	politician	90	TRUE	13/08/1926	25/11/2016
10	Antonin Scalia	lawyer	79	TRUE	11/03/1936	13/02/2016
11	Jo Cox	politician	41	TRUE	22/06/1974	16/06/2016
12	Janet Reno	lawyer	78	FALSE	21/07/1938	07/11/2016
13	Gwen Ifill	journalist	61	FALSE	29/09/1955	14/11/2016
14	John Glenn	astronaut	95	TRUE	28/07/1921	08/12/2016
15	Pat Summit	coach	64	TRUE	14/06/1952	28/06/2016
16	This					
17		has been really fun, but				
18	we're signing					
19			off		now!	

Importing data from spreadsheets

- Note how we only want to import the range `A5:F15` in the sheet called `other`. We can pass these values as the corresponding arguments to the `read_xlsx` function:

```r
library(readxl)
args(read_xlsx)
```

```
function (path, sheet = NULL, range = NULL, col_names = TRUE,
    col_types = NULL, na = "", trim_ws = TRUE, skip = 0, n_max = Inf,
    guess_max = min(1000, n_max), progress = readxl_progress(),
    .name_repair = "unique")
NULL
```

- Hence, to import this data, we should call:

```r
dir <- system.file(package = "readxl")
xlsx_filepath <- file.path(dir, "extdata", "deaths.xlsx")
head(as.data.frame(read_xlsx(xlsx_filepath, sheet = "other", range = "A5:F15")), 5)
```

```
            Name Profession Age Has kids Date of birth Date of death
1     Vera Rubin  scientist  88     TRUE    1928-07-23    2016-12-25
2    Mohamed Ali    athlete  74     TRUE    1942-01-17    2016-06-03
3   Morley Safer journalist  84     TRUE    1931-11-08    2016-05-19
4   Fidel Castro politician  90     TRUE    1926-08-13    2016-11-25
5 Antonin Scalia     lawyer  79     TRUE    1936-03-11    2016-02-13
```

Importing data from spreadsheets

- **Google Sheets** is another widely used spreadsheet program, which is free and web-based. Just like with Excel, in Google Sheets data are organized in worksheets (also called sheets) inside of spreadsheet files.
- Again, Base R does *not* have functionality to import data from Google Sheets spreadsheets. However, the package `googlesheets4` does. Its main function is `read_sheet`, which reads a Google Sheet from a URL or a file id.
- Given such a URL, its use is very similar to `read_xlsx`:

```r
library(googlesheets4)

gs4_deauth() # used to read publicly available Google Sheets
# Obtaining data on global life expectancy since 1800 from the Gapminder project
# For more information, see http://gapm.io/dlex
sheet_id <- "1RehxZjXd7_rG8v2pJYV6aY0J3LAsgUPDQnbY4dRdiSs"

df_gs <- as.data.frame(read_sheet(sheet_id,
                                  sheet = "data-for-world-by-year",
                                  range = "A1:D302"))
```

```
✔ Reading from "_GM-Life Expectancy- Dataset - v14".
✔ Range ''data-for-world-by-year'!A1:D302'.
```

Importing data from spreadsheets

Now, the data has successfully been imported:

```r
library(knitr)
kable(head(df_gs, 8)) # the function kable creates nice tables for presentations
```

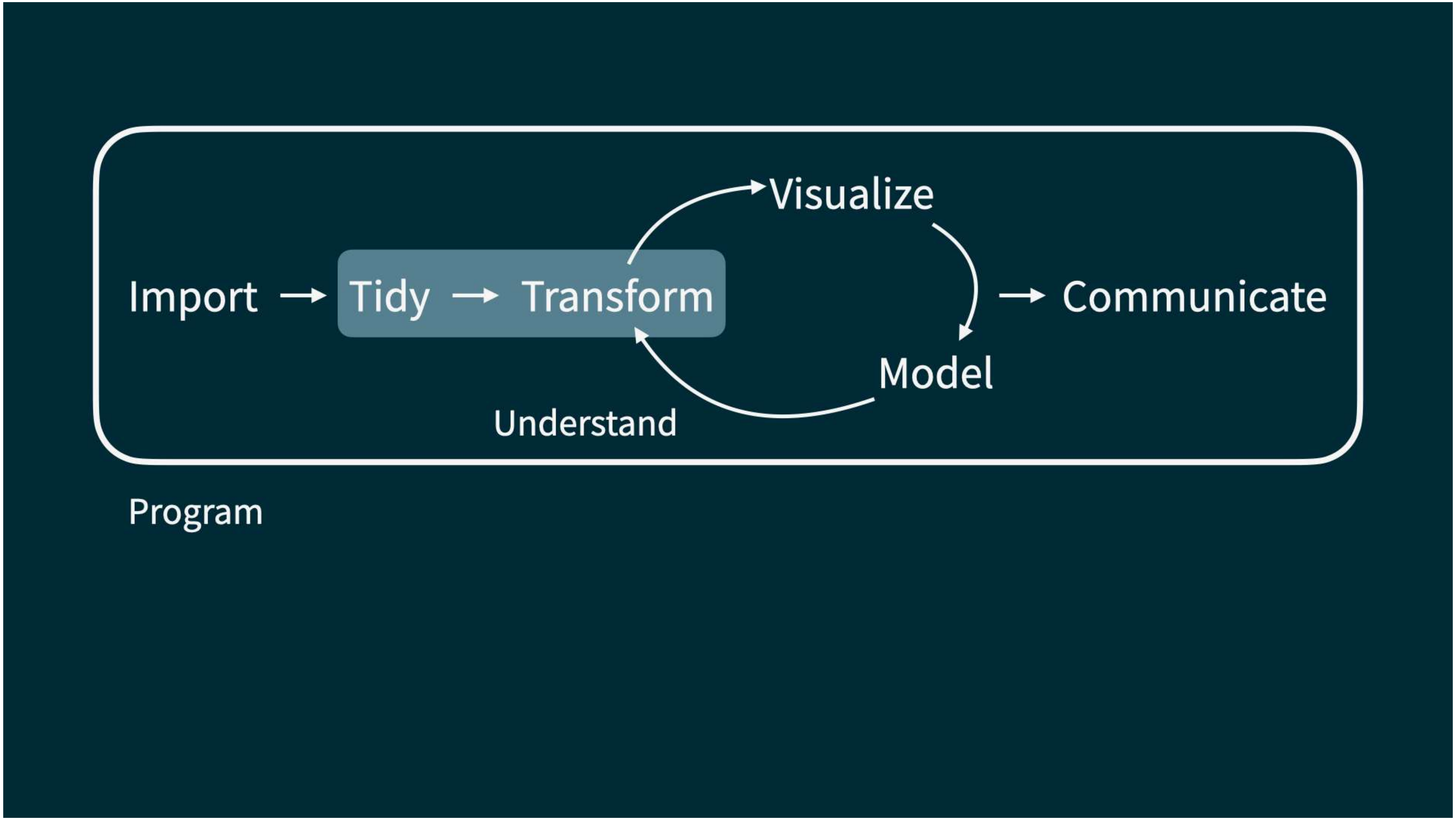
geo	name	time	Life expectancy
world	World	1800	30.64173
world	World	1801	30.71239
world	World	1802	30.60052
world	World	1803	30.27759
world	World	1804	30.19749
world	World	1805	30.78082
world	World	1806	30.79082
world	World	1807	30.73985

Importing data in other formats

- There is **a lot** of other data formats, which can be read into R with the help of other **packages**. The following provides a brief and inevitably incomplete overview:
  - Package `haven` for data from SPSS, Stata or SAS.
  - Package `DBI` along with a DBMS-specific backend allows you to run SQL queries on a data base and obtain the result as a `data.frame` directly in R.
  - Package `jsonline` for importing JSON files.
  - Package `xml2` for importing XML files.
  - …

Tidy and Transform

The Data Science workflow – Tidy and Transform



Tidy and Transform – Data wrangling

- After we imported data into R, we want to make it **easily processable** for visualizations or model building. This process could involve:
  - renaming columns to avoid confusion and unnecessary typing.
  - subsetting the data to use only parts of it, i.e. filtering.
  - handling incorrect and/or missing values.
  - aggregating the data to compute summary statistics.
  - reshaping the data to suit the needs of functions operating on them.
  - adding or replacing columns.
  - joining other data sets to enrich the information presented.
  - and much more…
- Jointly, we refer to these tidying and transformation tasks as **data wrangling**.

Tidy and Transform – Data wrangling

Example data set

Let's import some data that we will be using as an example:

```r
flights <- read.csv(file.path("data", "nyc13_flights.csv"))
head(flights)
```

```
  year month day actual.time.of.departure scheduled.time.of.departure
1 2013     1   1                      517                         515
2 2013     1   1                      533                         529
3 2013     1   1                      542                         540
4 2013     1   1                      544                         545
5 2013     1   1                      554                         600
6 2013     1   1                      554                         558
  depature.delay actual.time.of.arrival scheduled.time.of.arrival arrival.delay
1              2                    830                       819            11
2              4                    850                       830            20
3              2                    923                       850            33
4             -1                   1004                      1022           -18
5             -6                    812                       837           -25
6             -4                    740                       728            12
  carrier flight plane.tail.number origin destination air.time distance
1      UA   1545            N14228    EWR         IAH      227     1400
```

This data was adapted from the nycflights13 package. It contains information on all 166,158 domestic flights that departed from New York City (airports EWR, JFK and LGA) in the first six months of 2013. The data itself originates from the US Bureau of Transportation Statistics.

Column naming

The columns of this data set have quite long and descriptive names. Column names generally should:

- be as short as possible.
- be as long as necessary to be descriptive enough.
- not use spaces or special characters.
- only contain lowercase letters.
- use **snake_case** for multiple words.

These rules are good to keep in mind also when naming any other R object. Let's see what kind of information is contained in our example data set and how we could name the corresponding columns appropriately.

Column naming

Our example data set contains the following columns:

- `year`, `month`, `day`: date of departure
- `actual.time.of.departure`, `scheduled.time.of.departure`: actual and scheduled time of departure (in format HHMM or HMM). Too long, how about `dep_time` and `sched_dep_time`?
- `actual.time.of.arrival`, `scheduled.time.of.arrival`: actual and scheduled time of arrival (in format HHMM or HMM). Too long, how about `arr_time` and `sched_arr_time`?
- `departure.delay`, `arrival.delay`: departure and arrival delays, in minutes. To be consistent, how about `dep_delay` and `arr_delay`?
- `carrier`, `flight`: airline and flight number.
- `plane.tail.number`: plane tail number. Too long, how about `tailnum`?
- `origin`, `destination`: origin and destination airport. `origin` fine, but maybe `dest`?
- `air.time`: amount of time spent in the air, in minutes. Change to `air_time`?
- `distance`: distance between airports, in miles.
- `time.hour`: scheduled date and hour of the flight. Change to `time_hour`?

Column naming

To reset the column names of a `data.frame`, we can either use `colnames` or `names`:

```
head(names(flights))
```

```
[1] "year"                       "month"
[3] "day"                        "actual.time.of.departure"
[5] "scheduled.time.of.departure" "depature.delay"
```

```
names(flights) <- c("year", "month", "day", "dep_time", "sched_dep_time", "dep_delay",
                    "arr_time", "sched_arr_time", "arr_delay", "carrier", "flight",
                    "tail_num", "origin", "dest", "air_time", "distance", "time_hour")
head(flights)
```

```
  year month day dep_time sched_dep_time dep_delay arr_time sched_arr_time
1 2013     1   1      517            515         2      830            819
2 2013     1   1      533            529         4      850            830
3 2013     1   1      542            540         2      923            850
4 2013     1   1      544            545        -1     1004           1022
5 2013     1   1      554            600        -6      812            837
6 2013     1   1      554            558        -4      740            728
  arr_delay carrier flight tail_num origin dest air_time distance
1        11      UA   1545   N14228    EWR  IAH      227     1400
2        20      UA   1714   N24211    LGA  IAH      227     1416
3        33      AA   1141   N619AA    JFK  MIA      160     1089
4       -18      B6    725   N804JB    JFK  BQN      183     1576
5       -25      DL    461   N668DN    LGA  ATL      116      762
6        12      UA   1696   N39463    EWR  ORD      150      719
            time_hour
1 2013-01-01 05:00:00
```

Checking data types

Next, we should make sure that the data type and/or class used for each variable suits the data presented in this variable. In particular:

- Each numerical variable should of type `integer` or `double`.
- Each categorical variable should be a `factor`.
- Each non-categorical text-based variable should be of type `character`.
- Each date / date-time variable should be a `Date` or a `POSIXct`, respectively.

Let's have a look in our data set:

```
sapply(flights, typeof)
```

```
          year          month            day       dep_time sched_dep_time 
     "integer"      "integer"      "integer"      "integer"      "integer" 
     dep_delay       arr_time sched_arr_time      arr_delay        carrier 
     "integer"      "integer"      "integer"      "integer"    "character" 
        flight        tail_num         origin           dest       air_time 
     "integer"    "character"    "character"    "character"      "integer" 
      distance       time_hour 
     "integer"    "character" 
```

Adapting data types

Most of the data types in our data set seem appropriate, however we should:

- redefine `carrier`, `tail_num`, `origin` and `dest` to be a `factor`.
- redefine `time_hour` to be a `POSIXct` date-time.

Using the function `factor`, the first part should be no problem:

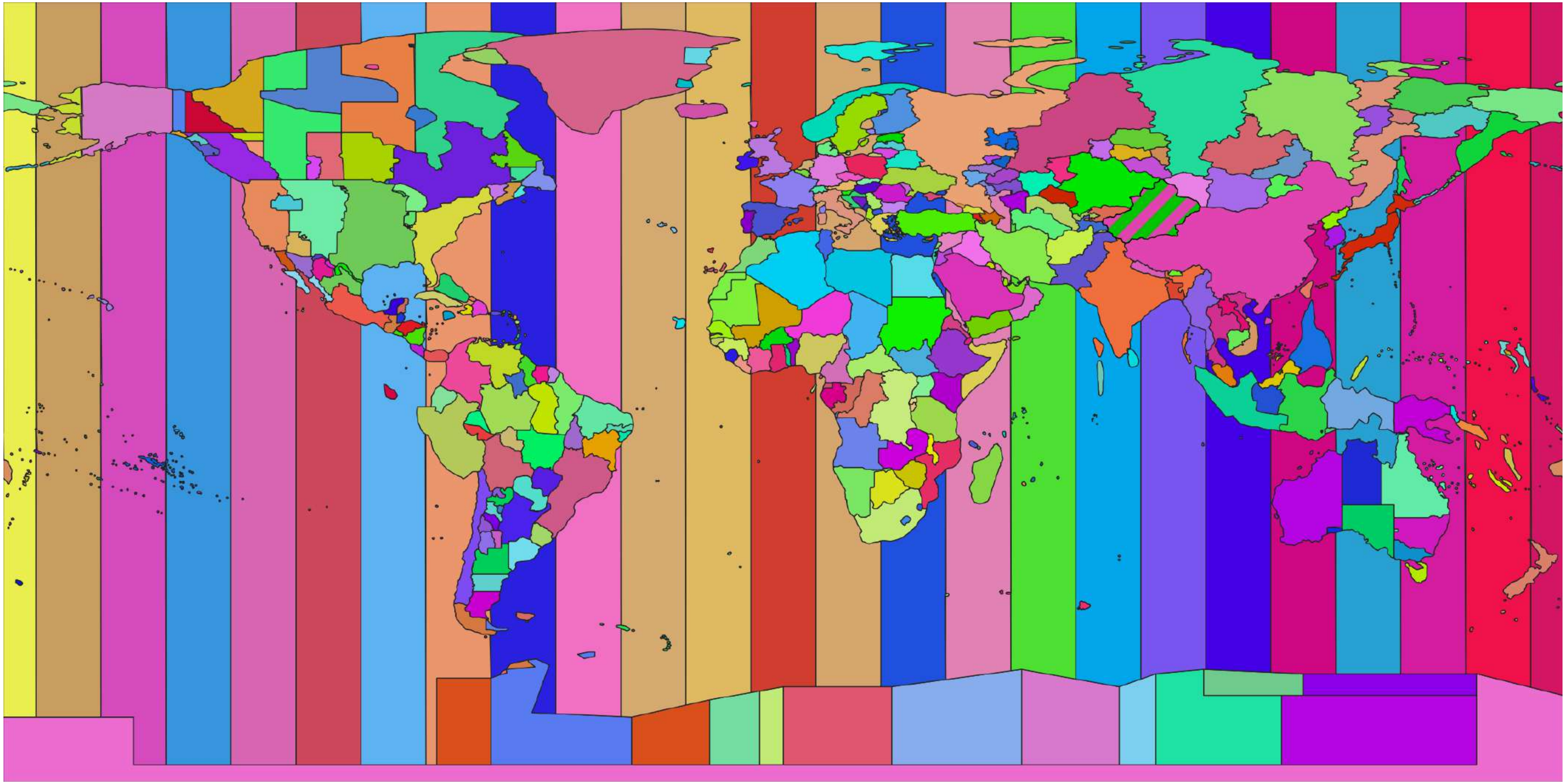
```r
factor_vars <- c("carrier", "tail_num", "origin", "dest")
for(var in factor_vars){
  flights[[var]] <- factor(flights[[var]])
  print(head(flights[[var]]))
}
```

```
[1] UA UA AA B6 DL UA
Levels: 9E AA AS B6 DL EV F9 FL HA MQ OO UA US VX WN YV
[1] N14228 N24211 N619AA N804JB N668DN N39463
3825 Levels: D942DN N0EGMQ N10156 N102UW N103US N104UW N10575 N105UW ... N9EAMQ
[1] EWR LGA JFK JFK LGA EWR
Levels: EWR JFK LGA
[1] IAH IAH MIA BQN ATL ORD
100 Levels: ABQ ACK ALB ATL AUS AVL BDL BGR BHM BNA BOS BQN BTV BUF BUR ... XNA
```

Date-times

- In the previous lecture, we have talked about **dates** in R, but we have not talked about how to represent **time**.
- As simple as time seem to be, representing time-related information with a computer can be incredibly complex. Think about:
  - time zones (changing in geographical composition over time),
  - daylight saving time (DST) and its relevance in different countries,
  - different formats for writing dates and times (e.g. 16:00 vs. 4:00 pm),
  - …
- In this course, we will fortunately stick to relatively simple cases. However, data in the real world does not always behave that nicely…

Date-times



This map partitions the world into regions where local clocks all show the same time and have done so since 1970. Talk about complexity…

UNIVERSITY OF SUSTAINABILITY
CHARLOTTE FRESENIUS PRIVATUNIVERSITÄT

Date-time in R – POSIXct

One way of representing date-time information in R is with the `POSIXct` class. Just as with dates, we can use **format strings** also to identify hour, minute, second, time zone, etc.

In our `flights` data set, the variable `time_hour` has a relatively simple format:

```
head(flights$time_hour)
```

```
[1] "2013-01-01 05:00:00" "2013-01-01 05:00:00" "2013-01-01 05:00:00"
[4] "2013-01-01 05:00:00" "2013-01-01 06:00:00" "2013-01-01 05:00:00"
```

Additionally, we know that these are all times from the NYC time zone. In R, we can make use of a data base of time zones with the help of the function `OlsonNames`, named after the original creator Arthur David Olson. In there, there is a time zone called **America/New_York**:

```
OlsonNames()[170]
```

```
[1] "America/New_York"
```

Date-time in R – POSIXct

Now, we can use the appropriate format string and time zone name to turn the `time_hour` variable into a `POSIXct` date-time. For this purpose, we require the function `as.POSIXct`:

```
flights$time_hour <- as.POSIXct(flights$time_hour,
                                tz = "America/New_York",
                                format = "%Y-%m-%d %H:%M:%S")
typeof(flights$time_hour)
```

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

```
class(flights$time_hour)
```

```
[1] "POSIXct" "POSIXt"
```

```
head(flights$time_hour, 2)
```

```
[1] "2013-01-01 05:00:00 EST" "2013-01-01 05:00:00 EST"
```

```
tail(flights$time_hour, 2)
```

```
[1] "2013-06-30 20:00:00 EDT" "2013-06-30 19:00:00 EDT"
```

Note how these `POSIXct` date-times now have `EST` (Eastern Standard Time) or `EDT` (Eastern Daylight Time) on them to indicate the time zone. `as.POSIXct` automatically applied the daylight saving time for the correct period.

Filtering

Now that we have the correct data types, we might want to **filter** our `data.frame`. **Filtering** refers to the process of keeping rows based on certain conditions imposed on the values of the columns. For example, we might want to find all flights on 14th February that were more than one hour delayed upon arrival at their destination.

We can filter a `data.frame` by **logical subsetting** of the rows. For this, we have to combine the conditions we want to impose by using the logical operators, `&` (**and**), `|` (**or**) and `!` (**not**). If we want to find the carriers and flight numbers of the aforementioned flights, we could do:

```r
head(flights[flights$month == 2 & flights$day == 14 & flights$arr_delay > 60,
             c("year", "month", "day", "carrier", "flight", "arr_delay")])
```

```
      year month day carrier flight arr_delay
38528 2013     2  14      9E   4023        92
38551 2013     2  14      DL    807       143
38604 2013     2  14      B6     56       118
38613 2013     2  14      B6    600        65
38623 2013     2  14      DL   1959       200
38624 2013     2  14      UA    517        95
```

Filtering

Let's see another example: suppose we want to find all flights that left from either Newark (EWR) or JFK to Los Angeles (LAX) on 14th February after 6:00 pm. We could do:

```
flights[flights$month == 2 &
          flights$day == 14 &
          (flights$origin == "EWR" | flights$origin == "JFK") &
          flights$dest == "LAX" &
          flights$dep_time > 1800,
        c("year", "month", "day", "carrier", "flight", "origin", "dest", "dep_time")]
```

```
      year month day carrier flight origin dest dep_time
39020 2013     2  14      AA    119    EWR  LAX     1812
39076 2013     2  14      DL     87    JFK  LAX     1906
39111 2013     2  14      AA     21    JFK  LAX     1943
39146 2013     2  14      VX    415    JFK  LAX     2017
39150 2013     2  14      UA    771    JFK  LAX     2027
39184 2013     2  14      DL   2363    JFK  LAX     2118
39185 2013     2  14      B6    677    JFK  LAX     2118
39201 2013     2  14      AA    185    JFK  LAX     2147
```

Note that – since there are only three possible `origin` airports in this data set – we could replace the third condition with

```
flights$origin != "LGA"
```

Filtering

That starts to be quite a lot of typing... To reduce the number of times that we have to type the name of the `data.frame`, we can use the `subset` function:

```
subset(flights,
       month == 2 & day == 14 & origin != "LGA" & dest == "LAX" & dep_time > 1800,
       c(year, month, day, carrier, flight, origin, dest, dep_time))
```

```
      year month day carrier flight origin dest dep_time
39020 2013     2  14      AA    119    EWR  LAX     1812
39076 2013     2  14      DL     87    JFK  LAX     1906
39111 2013     2  14      AA     21    JFK  LAX     1943
39146 2013     2  14      VX    415    JFK  LAX     2017
39150 2013     2  14      UA    771    JFK  LAX     2027
39184 2013     2  14      DL   2363    JFK  LAX     2118
39185 2013     2  14      B6    677    JFK  LAX     2118
39201 2013     2  14      AA    185    JFK  LAX     2147
```

This is much more clear and compact. Note that when using `subset`, the names of the columns we want to select do not have to be specified with quotation marks `""`.

Handling missing values

Let's see where in our data set we have **missing values** (indicated by NA):

```
n_missing <- sapply(flights, function(x) sum(is.na(x)))
n_missing[n_missing > 0]
```

```
 dep_time dep_delay  arr_time arr_delay  tail_num  air_time
     4883      4883      5101      5480      1521      5480
```

Only six variables seem to have missing values. Note how in this data set, the missing values can be **interpreted**:

- `dep_time` and `dep_delay` $\longrightarrow$ flight was **cancelled**. In these cases, `arr_time` and `arr_delay` are also NA.
- Additional missing values in `arr_time` $\longrightarrow$ flight was **diverted** to another destination airport.
- Additional missing values in `arr_delay` and `air_time`. Unknown reason for missingness, hard to construct without additional information.
- In some cases, the `tail_num` of the plane seems to be simply unknown.

Handling missing values

Let's start handling the cancelled flights first. Depending on the circumstances, we might want to add a variable to indicate a cancelled flight, like so…

```
flights$cancelled <- is.na(flights$dep_time)
head(flights[, c("year", "month", "day", "dep_time", "arr_time",
                 "carrier", "flight", "cancelled")], 5)
```

```
  year month day dep_time arr_time carrier flight cancelled
1 2013     1   1      517      830      UA   1545     FALSE
2 2013     1   1      533      850      UA   1714     FALSE
3 2013     1   1      542      923      AA   1141     FALSE
4 2013     1   1      544     1004      B6    725     FALSE
5 2013     1   1      554      812      DL    461     FALSE
```

… or remove cancelled flights from the data set all together and put them into their own `data.frame`. Let's go for that option:

```
cancelled_flights <- flights[is.na(flights$dep_time), ]
flights <- flights[!is.na(flights$dep_time), ]
head(cancelled_flights[, c("year", "month", "day", "dep_time", "arr_time",
                           "carrier", "flight", "cancelled")], 5)
```

```
     year month day dep_time arr_time carrier flight cancelled
839  2013     1   1       NA       NA      EV   4308      TRUE
840  2013     1   1       NA       NA      AA    791      TRUE
841  2013     1   1       NA       NA      AA   1925      TRUE
842  2013     1   1       NA       NA      B6    125      TRUE
1778 2013     1   2       NA       NA      EV   4352      TRUE
```

Handling missing values

Next, let's do the same also with diverted flights, which will contain all remaining flights that have `NA`s in the `arr_time` variable:

```
diverted_flights <- flights[is.na(flights$arr_time), ]
flights <- flights[!is.na(flights$arr_time), ]
head(diverted_flights[, c("year", "month", "day", "dep_time", "arr_time",
                         "carrier", "flight", "origin", "dest")], 5)
```

```
     year month day dep_time arr_time carrier flight origin dest
755  2013     1   1     2016       NA      EV   4204    EWR  OKC
1715 2013     1   2     2041       NA      B6    147    JFK  RSW
1757 2013     1   2     2145       NA      UA   1299    EWR  RSW
7040 2013     1   9      615       NA      9E   3856    JFK  ATL
7852 2013     1   9     2042       NA      B6    677    JFK  LAX
```

Let's look at how many missing values are remaining after separating out cancelled and diverted flights from our data set:

```
n_missing <- sapply(flights, function(x) sum(is.na(x)))
n_missing[n_missing > 0]
```

```
arr_delay  air_time
      379       379
```

Only a small number of `NA`s are remaining in the variables `arr_delay` and `air_time`. We will deal with them when we need to…

Descriptive statistics – Numeric variables

Now, we are finally ready to compute some descriptive statistics. Say, we want to know the average departure and arrival delay of a (not cancelled or diverted) flight. We use the function `mean`:

```
mean(flights$dep_delay)
```

```
[1] 13.65542
```

```
mean(flights$arr_delay)
```

```
[1] NA
```

The average departure delay is around 13.6 minutes, but the average arrival delay is `NA`? Why is that?

Of course, that’s exactly because of the remaining missing values in `arr_delay`. R cannot know the average of a vector of values where some values are unknown. However, most functions for descriptive statistics have an argument called `na.rm`:

```
mean(flights$arr_delay, na.rm = TRUE)
```

```
[1] 8.15129
```

Descriptive statistics – Numeric variables

Let's say we wanted to know what the maximum departure and arrival delays were. In that case, we would use the function `max` and also set its `na.rm` option to `TRUE` (strictly necessary only for `arr_delay`):

```
max(flights$dep_delay, na.rm = TRUE) / 60 # divide by 60 to get hours from minutes
```

```
[1] 21.68333
```

```
max(flights$arr_delay, na.rm = TRUE) / 60 # divide by 60 to get hours from minutes
```

```
[1] 21.2
```

So both maximum departure and arrival delays were over 21 hours!

Other functions for important univariate descriptive statistics of numeric variables are:

- `range` for both `min` and `max` in one go.
- `median` and `quantile` for quantiles.
- `var` and `sd` for variance and standard deviation.
- `fivenum` and `summary` for five-point summaries.

Descriptive statistics – Categorical variables

From categorical variables, we very often want to compute a **frequency table**. This can be achieved with the R function `table`. Let's say we want to know how many flights started from each of the three NYC airports:

```
# Absolute frequencies:
table(flights$origin)
```

```
  EWR   JFK   LGA 
58649 54097 48311 
```

```
# Relative frequencies:
prop.table(table(flights$origin))
```

```
      EWR       JFK       LGA 
0.3641506 0.3358873 0.2999621 
```

Or we want to know the distribution of carriers operating out of LaGuardia in %:

```
round(prop.table(table(flights$carrier[flights$origin == "LGA"]))*100, 2)
```

```
   9E    AA    AS    B6    DL    EV    F9    FL    HA    MQ    OO    UA    US 
 1.10 15.24  0.00  6.14 24.03  5.64  0.69  3.68  0.00 16.70  0.00  7.78 12.84 
   VX    WN    YV 
 0.00  5.70  0.46 
```

Descriptive statistics – Grouping

A very common action in data wrangling is computing statistics of a variable for each level of a categorical variable. In our `flights` data, we might be interested for example in:

- the average departure delay for each carrier,
- the median arrival delay in each month broken down by NYC airport,
- the fastest air time for each route,
- …

Such actions require us to **group** the data by the factor levels of the categorical variable and then compute statistics on the variable of interest for each of the resulting groups.

Descriptive statistics – Grouping

In R, this kind of action can be achieved with the functions `tapply` and `aggregate`. Let's first look at how `tapply` works for the three examples given on the previous slide:

```
# Average departure delay by carrier:
tapply(flights$dep_delay, flights$carrier, mean)
```

```
       9E        AA        AS        B6        DL        EV        F9        FL
18.238843  9.987883  8.058333 13.796756  9.530427 23.308381 24.197605 14.879078
       HA        MQ        OO        UA        US        VX        WN        YV
11.845304 11.580444 63.000000 12.404769  3.938624 14.777251 17.169844 22.549550
```

```
# Median arrival delay for each month and airport
tapply(flights$arr_delay, list(flights$origin, flights$month), median, na.rm = TRUE)
```

```
     1  2  3  4  5  6
EWR  0 -2 -4 -1 -6 -1
JFK -7 -5 -7 -4 -9 -1
LGA -4 -4 -7 -2 -9 -4
```

```
# Fastest air time on each route
head(tapply(flights$air_time, paste0(flights$origin, "_", flights$dest), min, na.rm = TRUE))
```

```
EWR_ALB EWR_ATL EWR_AUS EWR_AVL EWR_BDL EWR_BNA
     24      88     181      76      20      70
```

Descriptive statistics – Grouping

So `tapply` generally works like this:

```
tapply(variable_of_interest, list_of_grouping_variables, aggregation_function, ...)
```

By contrast, the function `aggregate` works like this:

```
aggregate(df_of_interest, list_of_grouping_variables, aggregation_function, ...)
```

`aggregate` then applies the `aggregation_function` to each variable in the `df_of_interest` by group.

So, we can compute both average departure and arrival delays by carrier in one go, for example:

```
head(aggregate(flights[,c("dep_delay", "arr_delay")],
               list(flights$carrier),
               mean, na.rm = TRUE))
```

```
  Group.1 dep_delay arr_delay
1      9E 18.238843  9.519527
2      AA  9.987883  1.475949
3      AS  8.058333 -3.036415
4      B6 13.796756 10.410469
5      DL  9.530427  1.745742
6      EV 23.308381 20.328515
```

Descriptive statistics – Grouping with `cut`

- A frequently encountered goal is to group not based on an existing categorical variable, but on different **intervals** of a numeric variable.
- For example, we might be interested in analyzing delay patterns based on **air time**: short-haul ($\leq$ 3 hours), medium-haul (3-6 hours) and long-haul (6-16 hours) (according to the IATA).
- To group by these flight length categories, we have to **cut** our `air_time` variable at these cut points. For this, we can use the function `cut`.
- Besides specifying the cut points, `cut` offers us also to **label** the levels of the resulting `factor`.

Descriptive statistics – Grouping with `cut`

So to add this variable to our `data.frame`, we might do:

```
flights$length_cat <- cut(flights$air_time, c(0, 180, 360, Inf),
                          labels = c("short-haul", "medium-haul", "long-haul"))

head(flights[, c("year", "month", "day", "dep_time", "arr_time",
                 "origin", "dest", "air_time", "length_cat")])
```

```
  year month day dep_time arr_time origin dest air_time  length_cat
1 2013     1   1      517      830    EWR  IAH      227 medium-haul
2 2013     1   1      533      850    LGA  IAH      227 medium-haul
3 2013     1   1      542      923    JFK  MIA      160  short-haul
4 2013     1   1      544     1004    JFK  BQN      183 medium-haul
5 2013     1   1      554      812    LGA  ATL      116  short-haul
6 2013     1   1      554      740    EWR  ORD      150  short-haul
```

Now, we can analyse average departure and arrival delays by these categories using `aggregate`, for instance:

```
aggregate(flights[,c("dep_delay", "arr_delay")],
          list(flights$length_cat), mean, na.rm = TRUE)
```

```
      Group.1 dep_delay arr_delay
1  short-haul  14.55357  10.06333
2 medium-haul  11.22840   2.39164
3   long-haul  10.17673  16.59834
```

Joining tables

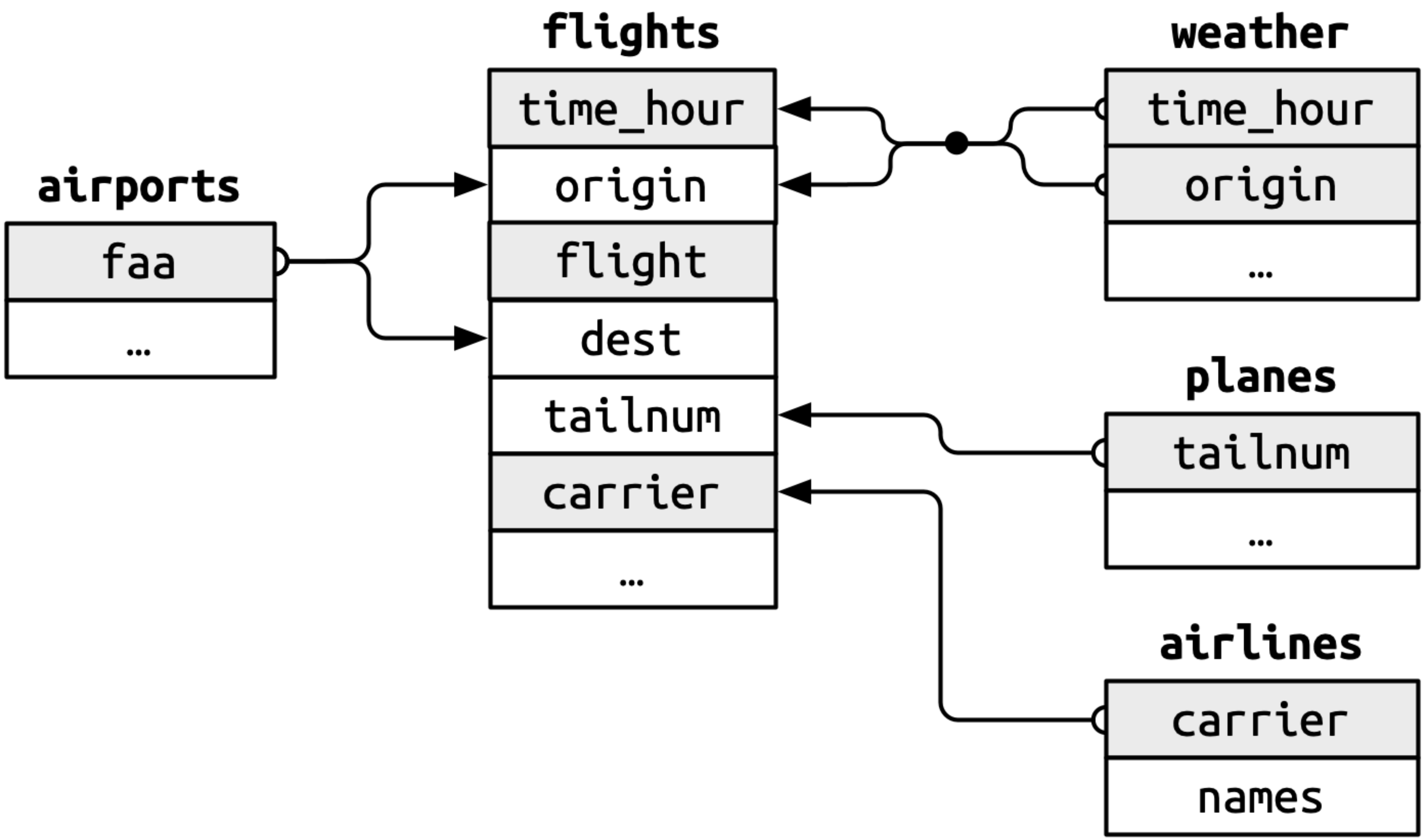
Our `flights` data set only contains codes for carrier, air plane and airports. To properly understand this data, we need to be able to **enrich** this data set by more information, such as:

- Full name of the carrier
- Full name and location of the origin and destination airports
- Type, model and size of the aircraft used
- Weather information at the origin airport at the time of departure

Indeed, any moderately complex data science project will involve **multiple tables** that must be **joined** together in order to answer the questions that you are interested in.

Joining tables

The nycflights13 package contains four additional tables that can be joined into the `flights` table. The below graph illustrates their relation:



Joining tables

Therefore, in the `flights` data set,

- the variables `origin` and `dest` are foreign keys that correspond to the primary key `faa` in `airports`.
- the variable `tail_num` is a foreign key that corresponds to the primary key `tail_num` in `planes`.
- the variable `carrier` is a foreign key that corresponds to the primary key `carrier` in `airlines`.
- the variables `origin` and `time_hour` constitute a **compound** foreign key that corresponds to the compound primary key constituted by `origin` and `time_hour` in `weather`.

To illustrate how we can **join** information from these tables into `flights` using keys, we start with the easiest example of `airlines`.

Joining tables

For this, we have to read in the `airlines` table and inspect it:

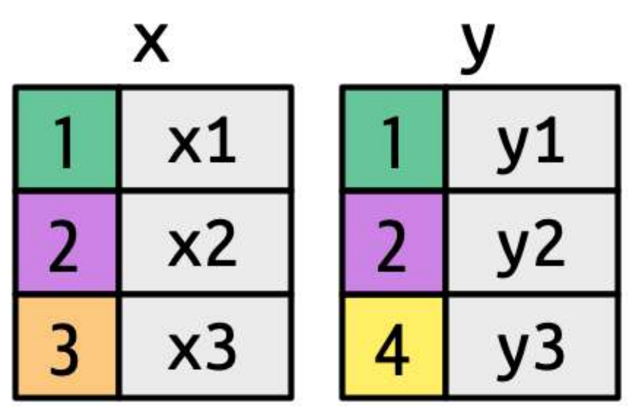
```r
airlines <- read.csv(file.path("data", "nyc13_airlines.csv"))
airlines
```

```
   carrier                        name
1       9E           Endeavor Air Inc.
2       AA      American Airlines Inc.
3       AS        Alaska Airlines Inc.
4       B6             JetBlue Airways
5       DL        Delta Air Lines Inc.
6       EV    ExpressJet Airlines Inc.
7       F9      Frontier Airlines Inc.
8       FL AirTran Airways Corporation
9       HA      Hawaiian Airlines Inc.
10      MQ                   Envoy Air
11      OO       SkyWest Airlines Inc.
12      UA       United Air Lines Inc.
13      US             US Airways Inc.
14      VX              Virgin America
15      WN      Southwest Airlines Co.
```

This is a very simple and small data set with the carrier code and name of 16 American airline companies. Now, how do we join this information into the `flights` data set?

Joining tables – Types of joins

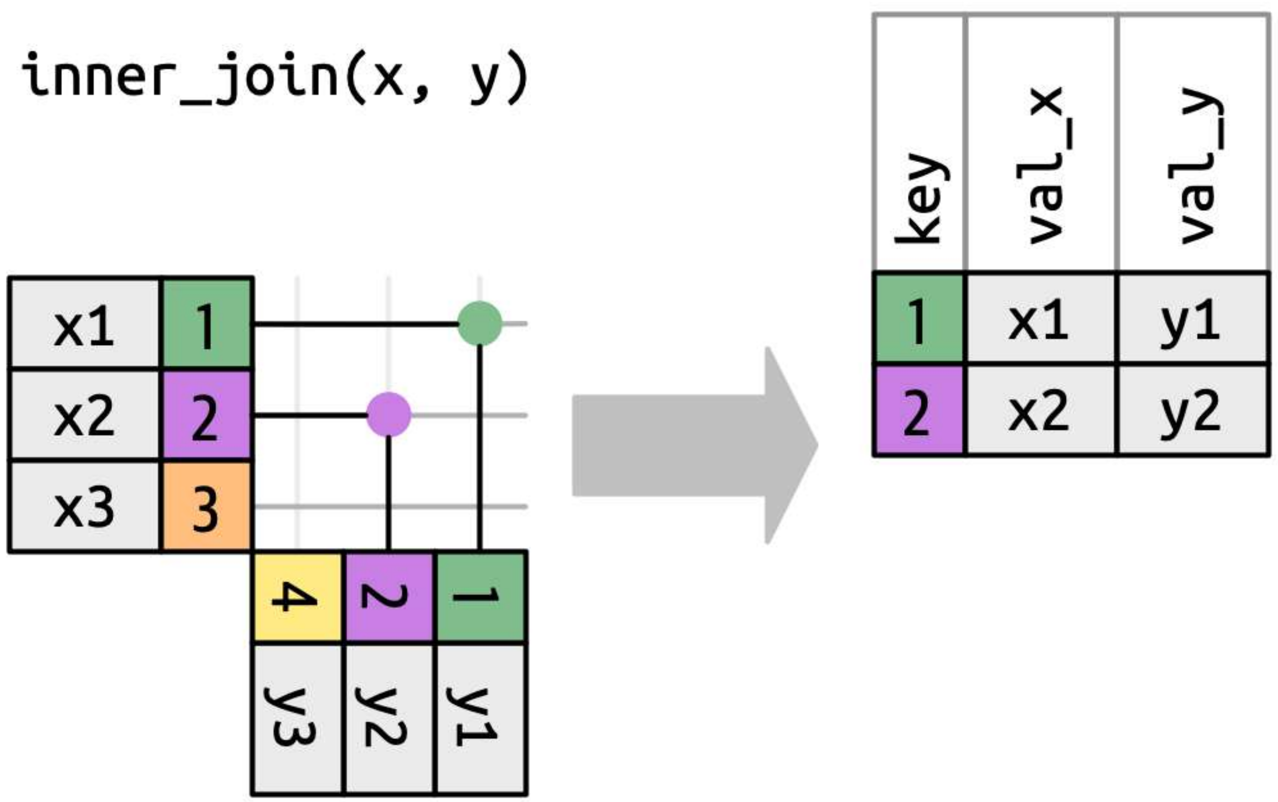
- There are many different **types of joins** that determine how two (or more) tables are brought together. We will illustrate only the most important ones with a toy example.
- Say, we have two tables, `x` and `y`, each with a `key` column and a column containing some values:



The colored key columns map background color to key value. The grey columns represent “value” columns.

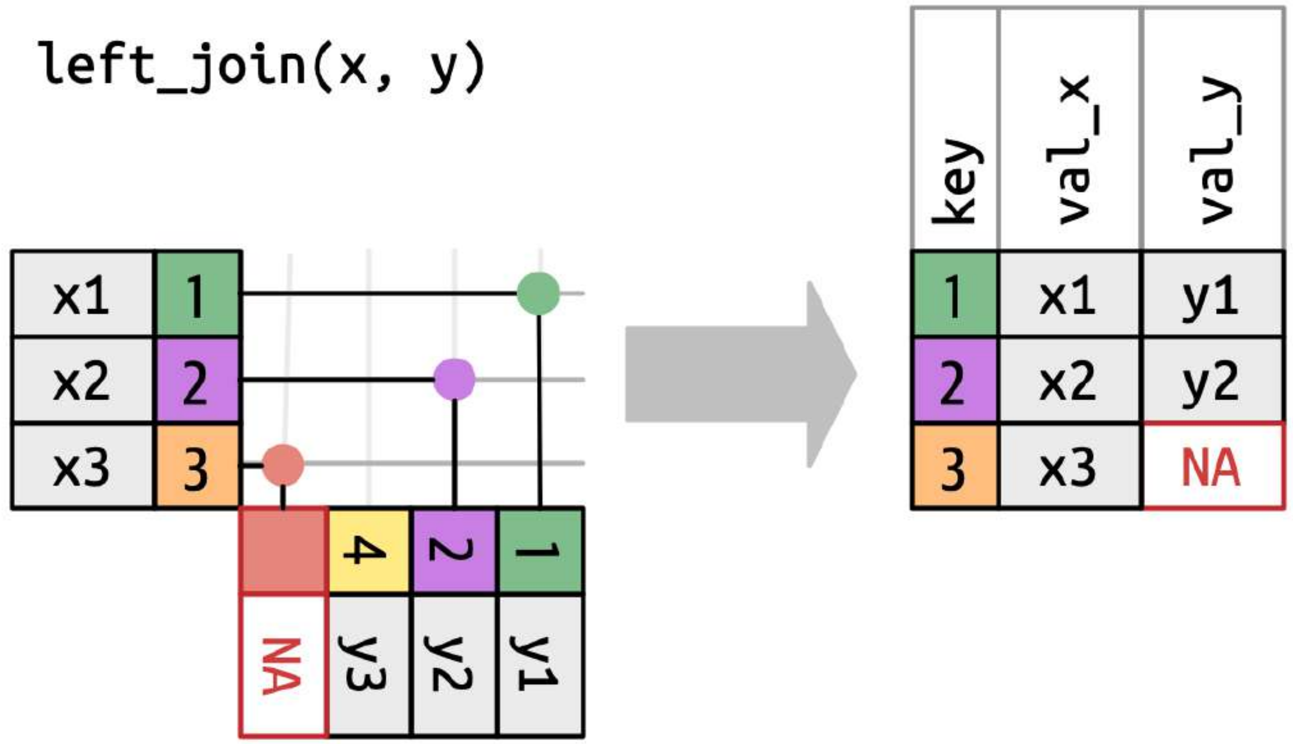
Joining tables – Inner join

- In an **inner join**, we want to keep only the rows that have information in both tables.
- In the example, this is only the case for data points with `key` 1 and 2. Therefore, the rows with `key` 3 and 4 do not make it to the joined table, if we join `x` and `y` on an inner join:



Joining tables – Left outer join

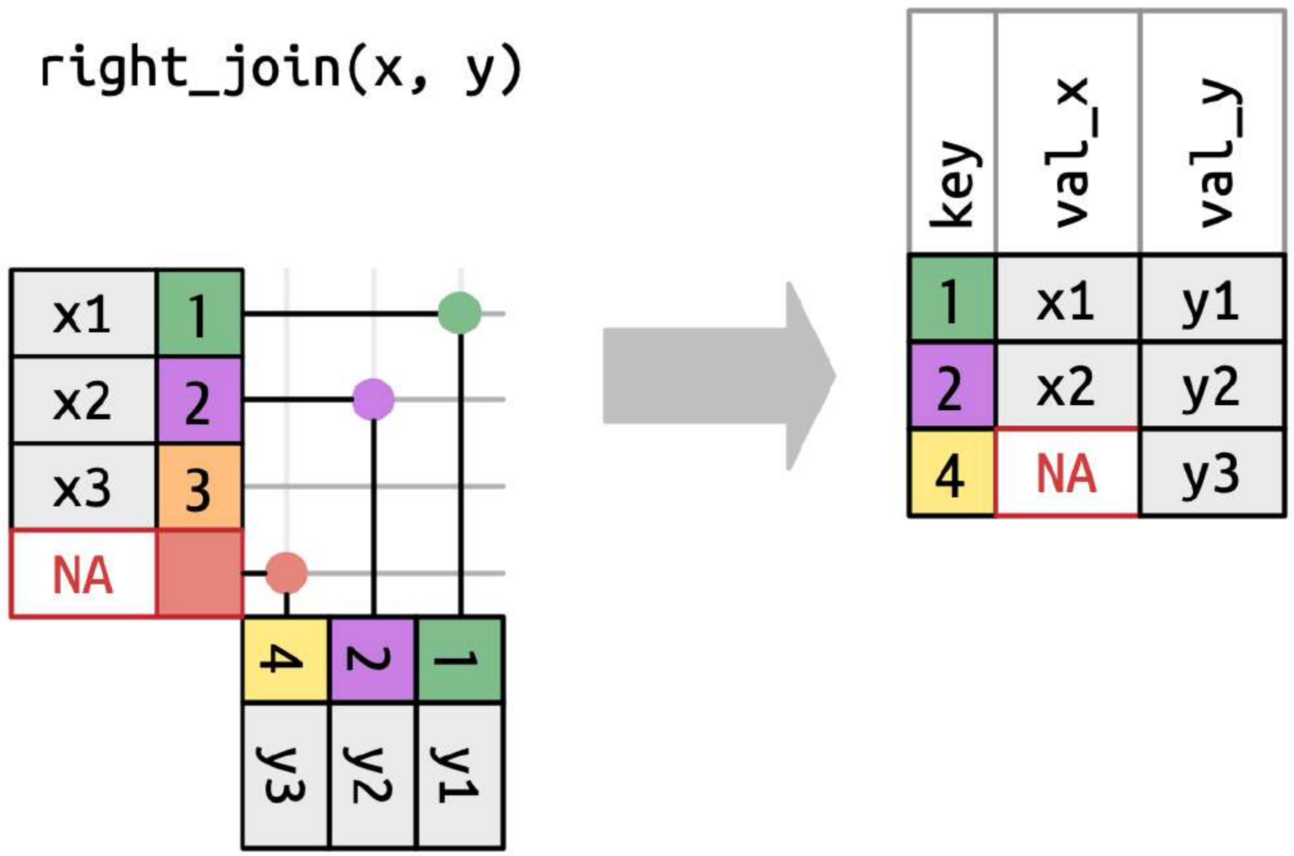
- In a **left outer join**, we want to keep all rows in the left table (in this case x). Rows without a matching key in y receive NA for the value column of y:



- For simplicity, such joins are usually simply called **left joins**.

Joining tables – Right outer join

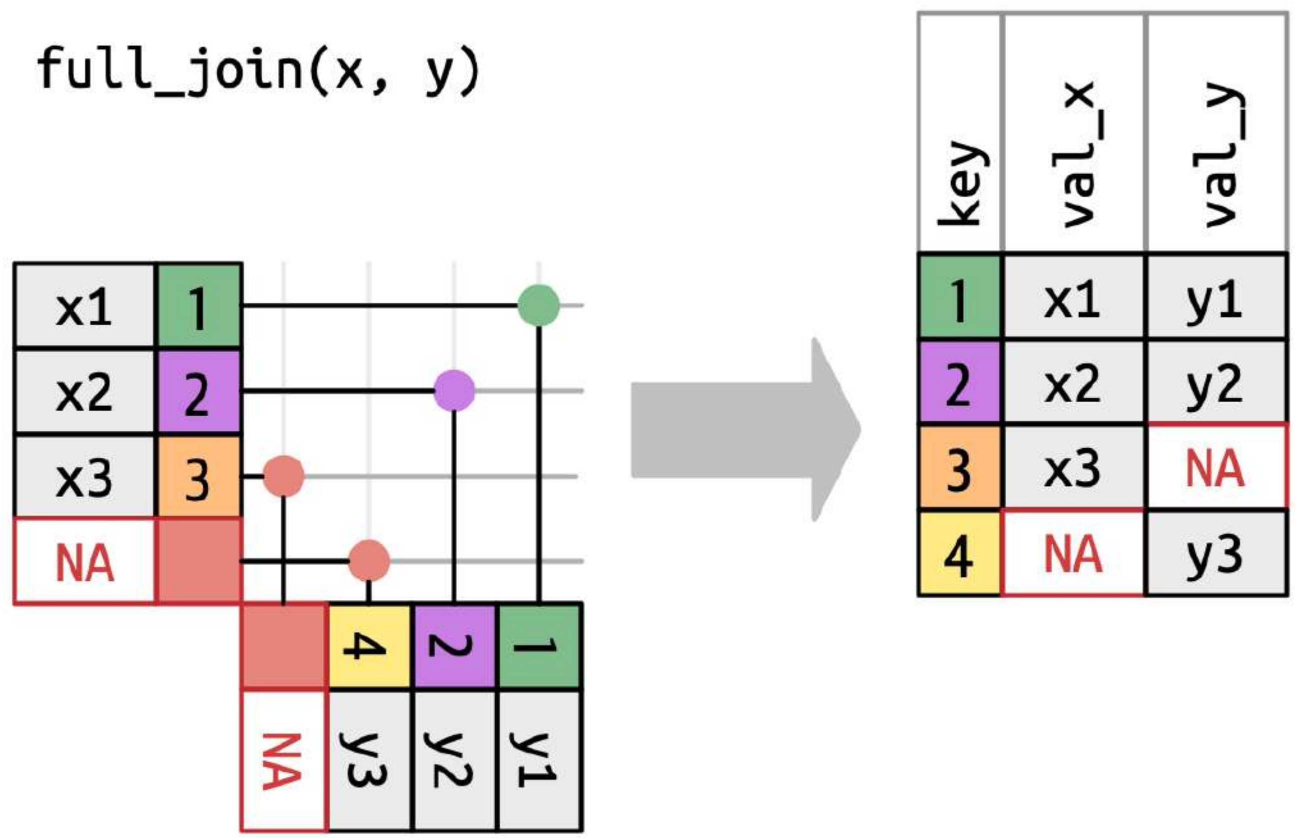
- In a **right outer join**, we want to keep all rows in the right table (in this case y). Rows without a matching key in x receive NA for the value column of x:



- For simplicity, such joins are usually simply called **right joins**.

Joining tables – Full outer join

- In a **full outer join**, we want to keep all the rows across both tables and fill the missing values with NAs:



- For simplicity, such joins are usually simply called **full joins**.

Performing joins with `merge`

In R, joining happens with the help of the function `merge`, whose arguments can be quite confusing at times… Here is the breakdown of the most important ones:

- `x` and `y` are the left and right `data.frame` to join.
- `by` is the name of the key(s) used for joining, if the column name of this key is the same in both `x` and `y`. If not, we specify the column names in `x` via the `by.x` argument and in `y` via the `by.y` argument.
- The arguments `all`, `all.x` and `all.y` specify the type of join:
  - Set `all` to `FALSE` for an **inner join**.
  - Set `all.x` to `TRUE` for a **left join**.
  - Set `all.y` to `TRUE` for a **right join**.
  - Set `all` to `TRUE` for a **full outer join**.

Performing joins with `merge`

So let's finally see joins in action! We **left join** the `airlines` data set into `flights` on the key of `carrier`. So, to do this in R, we run:

```
flights <- merge(flights, airlines, by = "carrier", all.x = TRUE)
head(flights[, c("year", "month", "day", "dep_time", "origin", "dest", "carrier", "name")])
```

```
  year month day dep_time origin dest carrier              name
1 2013     5  19     2034    LGA  TYS      9E Endeavor Air Inc.
2 2013     4  28     1621    JFK  MKE      9E Endeavor Air Inc.
3 2013     4  23      749    EWR  CVG      9E Endeavor Air Inc.
4 2013     6   5     2049    JFK  DCA      9E Endeavor Air Inc.
5 2013     1  19     1944    JFK  IAD      9E Endeavor Air Inc.
6 2013     6  27     1939    LGA  DAY      9E Endeavor Air Inc.
```

Note two things about this successful join:

- First, in the `airlines` data set, we found a match for every carrier in `flights`, so the left join did not produce any `NA`s. Proof:

```
sum(is.na(flights$name))
```

```
[1] 0
```

- Second, `merge` changed the order of the rows of the `data.frame`.

Tidy data

- To finish this chapter, let’s talk about **tidy data**.
- Rather than just meaning “clean”, tidy data actually refers to a specific way of organizing your data that is beneficial for the types of actions we want to perform on them. There are three interrelated rules that make data tidy:
  - Each variable is a column; each column is a variable.
  - Each observation is a row; each row is an observation.
  - Each value is a cell; each cell is a single value.
- These rules might seem pretty obvious, but actually, most real-world data sets do not meet these requirements as data is often organized to facilitate some goal other than analysis.

Tidy data

- To illustrate this point, we will have a look a new data set. It contains the fertility rates in Germany and South Korea in the years 1960 to 2015:

```r
fertility <- read.csv(file.path("data", "fertility.csv"))
fertility[, 1:12]
```

```
      country X1960 X1961 X1962 X1963 X1964 X1965 X1966 X1967 X1968 X1969 X1970
1     Germany  2.41  2.44  2.47  2.49  2.49  2.48  2.44  2.37  2.28  2.17  2.04
2 South Korea  6.16  5.99  5.79  5.57  5.36  5.16  4.99  4.85  4.73  4.62  4.53
```

```r
dim(fertility)
```

```
[1]  2 57
```

- This data is **untidy**. Why?
  - It contains a variable (namely the year) in **column names**, but according to the principles of tidy data, each variable should be its own column.
  - The observations are fertility rates in two countries, so these values should be organized in **rows**.

Tidy data

Due to its shape, such data is said to be in a **wide format** (few rows, lots of columns). We want to reshape it to **long format** (lots of rows, few columns). For this purpose, we use the R function `reshape`:

```
fertility_long <- reshape(fertility, direction = "long",
                          varying = list(names(fertility)[-1]), v.names = "fertility_rate",
                          idvar = "country",
                          timevar = "year", times = 1960:2015)
rownames(fertility_long) <- NULL
head(fertility_long, 8)
```

```
      country year fertility_rate
1     Germany 1960           2.41
2 South Korea 1960           6.16
3     Germany 1961           2.44
4 South Korea 1961           5.99
5     Germany 1962           2.47
6 South Korea 1962           5.79
7     Germany 1963           2.49
8 South Korea 1963           5.57
```

Now, the data is **tidy**! Note that it is exactly the same underlying data, just represented in a slightly different way.

Tidy data

The function `reshape` takes some practice to get used to. But once we know how to get our data **tidy**, there are multiple advantages to using this consistent way of organizing your data:

- As we will see, many functions for data analysis in R cannot be used, unless the data is tidy. This applies in particular to the visualizations we will create with the `ggplot2` package.
- Consistent data structures are easier to work with. If every data set "looks and feels" the same, you can build routines that will make your analyses more efficient and effective.
- Having variables consistently in columns is particularly sensible in R due to its **vectorized nature**. R performs at its best when it is able to run functions on vectors of values. With a tidy data format, we can fully leverage this potential.