

R for social science and business analytics

Raffaele Vacca

2024-04-09

Contents

1	Overview and setup	5
1.1	Workshop setup	5
1.2	Workshop materials	6
1.3	R settings	6
1.4	Data	7
1.5	Author and contacts	7
2	Introduction to R	9
2.1	Starting R and loading packages	9
2.2	Objects in R	12
2.3	Arithmetic, statistical, and relational operations	20
2.4	Subsetting	25
2.5	Pipes and the <code> ></code> operator	32
2.6	Writing your own R functions	33
2.7	Types and classes of objects	36
3	Data wrangling and descriptive statistics	41
3.1	Subsetting and renaming variables	42
3.2	Recoding, cleaning, creating variables	48
3.3	Filtering and rearranging cases	55
3.4	Describing or summarizing the data	60
4	Data visualization	67
5	Creating reproducible reports	69

Chapter 1

Overview and setup

This is a series of four workshop sessions about R programming for social science research and business analytics:

1. Introduction to R (`01_basics.R` script): R objects, vectors and matrices, arithmetic and logical operations, subsetting and indexing, data frames and lists, R functions.
2. Data wrangling and descriptive statistics (`02_wrangling.R` script): importing data, subsetting, ordering cases and variables, transforming and recoding, joining and appending data frames; frequency tables and crosstabs, mean, standard deviation and other descriptive functions, descriptive statistics for data subsets.
3. Data visualization (`03_visualization.R` script): the `ggplot2` package and the grammar of graphics, geometries and aesthetics, visualizing univariate distributions (histograms, boxplots, simple bar plots etc.), Visualizing associations between two or more variables (scatterplots, complex bar plots, etc.).
4. Creating reproducible reports (`04_reports.R` script): reproducible reports in different formats, RMarkdown basics, R code chunks, chunk options, inline R code.

1.1 Workshop setup

To take this workshop you need to:

1. Download the last version of **R** [here](#)
 - Select a location near you in the web page above
 - Follow instructions to install R on your computer
2. Download **RStudio** (free version) [here](#)
 - Follow instructions to install RStudio on your computer
3. Bring your **laptop** to the workshop

4. Download the **workshop folder** and save it to your computer: see below
 - I recommend that you do this in class at the beginning of the workshop so as to download the most updated version of the folder.
5. Once in class, go to the workshop folder on your computer (point 4 above) and double-click on the **R project** file in it (**.Rproj** extension).
 - That will open RStudio: you're all set!

NOTE: It's very important that you save the workshop folder *as downloaded* to a location in your computer, and open the **.Rproj** *within that folder*. By doing so, you will be opening RStudio *and* setting the workshop folder as your R working directory. All our R scripts assume the workshop folder is your working directory. You can type `getwd()` in your R console to see the path to your R working directory and make sure that it's correctly pointing to the location of the workshop folder in your computer.

1.2 Workshop materials

The materials for this workshop consist of this website and the workshop folder.

You can **download** the workshop folder from this GitHub repository:

1. Click on the *Code* green button > Download ZIP
2. Unzip the folder and save it to your computer

The workshop folder contains several files and subfolders, but you only need to focus on the following:

- **scripts** subfolder: all the R code shown in this website.
- **data** subfolder: all the data we're going to use.
- **r-social-business-analytics.Rproj**: the workshop's R project file (you use this to launch RStudio).

The **scripts** subfolder includes different R script (.R) files. You can access and run the R code in each script by opening that .R file in RStudio.

1.3 R settings

1.3.1 Required R packages

We'll install and load these in class:

- **janitor**
- **skimr**
- **tidyverse**

The tidyverse isn't a single package, it's a collection of packages that share a common set of functions and principles, including **dplyr**, **ggplot2**, and **purrr**. See the tidyverse website for more information.

1.3.2 RStudio options

RStudio gives you the ability to select and change various settings and features of its interface: see the **Preferences...** menu option.

These are some of the settings you should pay attention to:

- **Preferences... > Code > Editing > Soft-wrap R source file.**
Here you can decide whether or not to wrap long code lines in the editor. When code lines in a script are *not* wrapped, some code will be hidden if script lines are longer than your editor window's width (you'll have to scroll right to see the rest of the code). With a script open in the editor, try both options (checked and unchecked) to see what you're more comfortable with.
- **Preferences... > Code > Display > Highlight R function calls.**
This allows you to highlight all pieces of code that call an R function ("command"). I find function highlights very helpful to navigate a script and suggest that you check this option.

1.4 Data

This workshop uses an anonymized subset of the survey data collected for Valore D's Oltre le generazioni study, limited to 1000 randomly selected cases (survey respondents, i.e. employees) and with fictitious company names. All data are in the **data** subfolder.

1.5 Author and contacts

I'm an assistant professor of sociology at the University of Milan in the Department of Social and Political Sciences and its Behave Lab. My main research and teaching interests are social networks, migration, health inequalities, and studies of science. I also teach and do research on data science, statistics, and computational methods for the social sciences. More information about me, my work and my contact details is [here](#).

Chapter 2

Introduction to R

The script covers the following topics:

- Starting R, getting help with R.
- Creating and saving R objects.
- Vectors and matrices, data frames and tibbles.
- Arithmetic and relational operations.
- Subsetting vectors, matrices, and data frames.
- Pipes and the pipe operator.
- Object types and classes.
- Writing R functions.

2.1 Starting R and loading packages

- Before starting any work in R, you normally want to do two things:
 - Make sure your R session is pointing to the correct *working directory*.
 - Install and/or load the *packages* you are going to use.
- **Working directory.** By default, R will look for files and save new files in this directory.
 - Type `getwd()` in the console to view your current working directory.
 - If you opened RStudio by double-clicking on a project (`.Rproj`) file, then the working directory is the folder where that file is located.
 - You can always use `setwd()` to manually change your working directory to any path, but it's usually more convenient to work with R projects and their default working directory instead.
 - In RStudio, you can also check the current working directory by clicking on the **Files** panel.
- **R packages.** There are two steps to using a package in R:
 1. **Install** the package. *You do this just once.* Use `install.packages("package_name")` or the appropriate RStudio menu (**Tools > Install Packages...**).

Once you install a package, the package files are in your system R folder and R will be able to always find the package there.

2. **Load** the package in your current session. Use `library(package_name)` (no quotation marks around the package name). *You do this in each R session in which you need the package*, that is, every time you start R and you need the package.
- An R package is just a collection of **functions**. You can only use an R function if that function is included in a package you loaded in the current session.
 - Sometimes two different functions from two different packages have the same **name**. For example, both the **igraph** package and the **sna** package have a function called **degree**. If both packages are loaded, typing just **degree** might give you unexpected results, because R will pick one of the two functions (the one in the package that was loaded most recently), which might not be the function you meant.
 - To avoid this problem, you can use the `package::function()` notation: `igraph::degree()` will always call the **degree** function from the **igraph** package, while `sna::degree()` will call the **degree** function from the **sna** package.
 - Tip: To check the package that a function comes from, just go to that function's manual page. The package will be indicated in the first line of the page. E.g., type `?degree` to see where the **degree** function comes from.
 - If no currently loaded package has a function called **degree**, then typing `?degree` will produce a warning (No documentation for 'degree').
 - If multiple, currently loaded packages have a function called **degree**, then typing `?degree` will bring up a page with the list of all those packages.
 - This workshop will use different packages, listed here.

2.1.1 Console vs scripts

- When you open RStudio, you typically see two separate panels: the script editor and the console. You can write R code in either of them.
- **Console**. Here you write R code line by line. Once you type a line, you press ENTER to execute it. By pressing ARROW UP you go back to the last line you ran. By continuing to press ARROW UP, you can navigate through all the lines of code you previously executed. This is called the “commands history” (all the lines of code executed in the current session). You will lose all this code (all the history) when you quit R, unless you explicitly save the history to a file (which is not what you typically do, you should just write the code in a script).
- **Script editor**. Here you write a script. This is the most common way of working with R. A script is simply a plain text file where all your R code

is saved. If your work is in a script, it is **reproducible**.

- Both the R standard GUI and RStudio have a script editor with several helpful tools. Among other things, these allow you to run a script while you write it. By pressing **CTRL+ENTER** (Windows) or **CMD+ENTER** (Mac), you run the script line your cursor is on (or the selected script region).
 - Note that with RStudio you can run the single script line where your cursor is; a whole highlighted region of code; the region of code from the beginning of the script up to the line where your cursor is; the region of code from the line where your cursor is up to the end of the script. See the *Code* menu and its keyboard shortcuts.
- The script editor also allows you to save your script. In RStudio, see **File > Save** and its keyboard shortcut. R script files commonly have a `.R` extension (e.g. “`myscript.R`”). But note that a script file is just a text file (like any `.txt` file), which you can open and edit in any text editor, or in Microsoft Word and the likes.
- You can also run a whole script altogether — this is called **sourcing** a script. By running `source("myscript.R")`, you source the script file `myscript.R` (assuming the file is in your working directory, otherwise you’ll have to enter the whole file path). In RStudio: see *Code > Source* and its keyboard shortcut.
- In both the console and the script editor, any line that starts by `#` is called a **comment**. R disregards comments — it just prints them as they are in the console (does not parse and execute them as programming code). Remember to always use comments to document what your code is doing (this is good for yourself and for others).
- In RStudio you can navigate the script headings in your script with a drop-down menu in the bottom-left of the script editor. Any line that starts by `#` and ends by `####`, `----`, or `=====` is read as a heading by RStudio.

2.1.2 Getting help

- Getting help is one of the most common things you do when using R. As a beginner, you’ll constantly need to get help (for example, read manual pages) about R functions. Also as an experienced user, you’ll often need to go back to the manual pages of particular functions or other R help resources. At any experience level, using R involves constantly using its documentation and help resources.
- The following are a few help tools in R:
 - `help(...)` or `?...` are the most common ways of getting help: they send you to the R manual page for a specific function. E.g. `help(sum)` or `?sum` (they are equivalent).
 - `help.start()` (or RStudio: **Help > R Help**) gives you general help pages in html (introduction to R, references to all functions in all installed packages, etc.).
 - `demo()` gives you demos on specific topics. Run `demo()` to see all available topics.

- `example()` gives you example code on specific functions, e.g. `example(sum)` for the function `sum`.
- `help.search(...)` or `??...` search for a specific string in the manual pages, e.g. `??histogram`.
- In addition to built-in help facilities within R, there are plenty of ways to get **R help online**. Certain popular R packages have their own website, for example `ggplot2` and `igraph`. Other websites for general R help include `rdocumentation.org` and `stackoverflow.com`. See the workshop slides or talk to me for more information.

```
# What's the current working directory?
# getwd()
# Un-comment to check your actual working directory.

# Change the working directory.
# setwd("/my/working/directory")
# (Delete the leading "#" and type in your actual working directory's path
# instead of "/my/working/directory")

# You should use R projects (.Rproj) to point to a working directory instead of
# manually changing it.

# Suppose that we want to use the package "igraph" in the following code.
library(igraph)

# Note that we can only load a package if we have it installed. In this case, I
# have igraph already installed. Had this not been the case, I would have have
# needed to install it:
# install.packages("igraph").
# (Packages can also be installed through an RStudio menu item).

# Let's load another suite of packages we'll use in the rest of this script.
library(tidyverse)

# Note that whatever is typed after the "#" sign has no effect but to be printed
# as is in the console: it is a comment.
```

2.2 Objects in R

In R, everything that **exists** is an object. Everything that **happens** is a function call. - John Chambers

- R is an object-oriented programming language. Everything is contained in an **object**, including data, analysis tools and analysis results. Things such as datasets, commands (called “functions” in R), regression results,

descriptive statistics, etc., are all objects.

- An object has a *name* and a *value*. You create an object by **assigning** a value to a name.
 - You assign with `<-` or with `=`.
 - R is **case-sensitive**: the object named `mydata` is different from the object named `Mydata`.
- Whenever you run an operation or execute a function in R, you need to assign the result to an object if you want to save it and re-use it later. **Assign it or lose it**: anything that is not assigned to an object is just printed to the console and lost.
- Objects have a size (bytes, megabytes, etc.) and a **type** (technically, a *class*, a *type* and a *mode* — more on this later).
- A **function** is a particular type of object. Functions take other objects as arguments (input) and return more objects as a result (output). R functions are what other data analysis programs call “commands”. See Section 2.6 for more about functions.

2.2.1 The workspace

- During your R session, objects (data, results) are located in the computer’s main memory. They make up your workspace: the set of **all the objects** currently in memory. They will disappear when you quit R, unless you save them to files on disk.
- What’s in your current workspace?
 - The function `ls()` shows you a full list of the objects currently in the workspace.
 - Alternatively, in RStudio open the Environment panel to get a clickable list of objects currently in the workspace (if you don’t see your Environment panel, check **Preferences... > Pane Layout**).

2.2.2 Saving and removing objects

- Two main functions to **save** R objects to files: `save()` (saves specific objects, its arguments); `save.image()` (saves all the current workspace).
- Unless you specify a different path, all files you save from R are put in your current working directory.
- The most common file extensions for files that store R objects are `.rda` and `.RData`.
- If you have a file with R objects, say `objects.rda`, you can **load** it in your current R session using the `load()` function: `load(file="objects.rda")`. This assumes `objects.rda` is in your current working directory (otherwise you’ll have to specify the whole file path).
- The function `rm()` **removes** specific objects from the workspace. You can use it to clear the workspace from all existing objects by typing `rm(list=ls())` (remember that `ls()` returns a character vector with the names of all the objects in the current workspace).

```
# Create the object a: assign the value 50 to the name "a"
a <- 50

# Display ("print") the object a.
a

## [1] 50

# Let's create another object.
b <- "Mark"

# Display it.
b

## [1] "Mark"

# Create and display object at the same time.
(obj <- 10)

## [1] 10

# Let's reuse the object we created for a simple operation.
a + 3

## [1] 53

# What if we want to save this result?
result <- a + 3

# All objects in the workspace
ls()

## [1] "a"      "b"      "obj"    "result"

# Now we can view that result whenever we need it.
result

## [1] 53

# ...and further re-use it
result*2

## [1] 106

# Note that R is case-sensitive, "result" is different from "reSult".
reSult

## Error in eval(expr, envir, enclos): object 'reSult' not found

# Let's clear the workspace before proceeding.
rm(list=ls())
```

```
# The workspace is now empty.
ls()
```

```
## character(0)
```

2.2.3 Vector and matrix objects

- **Vectors** are the most basic objects you use in R. Vectors can be numeric (numerical data), logical (TRUE/FALSE data), or character (string data).
- The basic function to create a vector is `c` (**concatenate**).
- Other useful functions to create vectors: `rep` and `seq`.
 - Another function we'll use to create vectors later in the workshop is `seq_along`. `seq_along(x)` creates a vector consisting of a sequence of integers from 1 to `length(x)` in steps of 1.
 - Also keep in mind the `:` shortcut: `c(1, 2, 3, 4)` is the same as `1:4`.
- The **length** (number of elements) is a basic property of vectors: `length(x)` returns the length of vector `x`.
- When we **print** vectors, the numbers in square brackets indicate the positions of vector elements.
- To create a matrix: `matrix`. Its main arguments are the cell values (within `c()`), number of rows (`nrow`) and number of columns (`ncol`). Values are arranged in a `nrow` x `ncol` matrix *by column*. See `?matrix`.
- When we **print** matrices, the numbers in square brackets indicate the row and column numbers.

```
# Let's create a simple vector.
(x <- c(1, 2, 3, 4))
```

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

```
# Shortcut for the same thing.
(y <- 1:4)
```

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

```
# What's the length of x?
length(x)
```

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

```
# Note that when we print vectors, numbers in square brackets indicate positions
# of the vector elements.
```

```
# Create a simple matrix.
adj <- matrix(c(0,1,0, 1,0,0, 1,1,0), nrow= 3, ncol=3)
```

```
# This is what our matrix looks like:
adj
```

```
##      [,1] [,2] [,3]
## [1,]    0    1    1
## [2,]    1    0    1
## [3,]    0    0    0
```

Notice the row and column numbers in square brackets.

2.2.4 Data frames

- “Data frame” is R’s name for dataset. A dataset is a collection of cases (rows), and variables (columns) which are measured on those cases.
- When **printed** in R, data frames look like matrices. However, unlike matrix columns, data frame columns can be of different types, e.g. a numeric variable and a character variable.
- On the other hand, just like matrix columns, data frame columns (variables) must all have the same **length** (number of cases). You can’t put together variables (vectors) of different length in the same data frame.
- Although data frames look like matrices, in R’s mind they are a specific kind of *list*. In fact, the **class** of a data frame is **data.frame**, but the **type** of a data frame is **list**. The list elements for a data frame are its variables (columns).
- **Tibbles.** The tidyverse packages, which we use in this workshop, rely on a more efficient form of data frame, called *tibble*.
 - A tibble has class **tbl_df** and **data.frame**. This means that, to R, a tibble is *also* a data frame, and any function that works on data frames normally also works on tibbles.
 - A tibble has a number of advantages over a traditional data frame, some of which we’ll see in this workshop.
 - One of the advantages is the clearer and more informative way in which tibbles are printed. When we print a tibble data frame we can immediately see its number of rows, number of columns, names of variables, and type of each variable (numeric, integer, character, etc.).
 - To convert an existing data frame to tibble: **as_tibble**. To create a tibble from scratch (similar to the **data.frame** function in base R): **tibble**.
- While data frames can be created manually in R (with the functions **data.frame** in base R and **tibble** in **tidyverse**), data are most commonly imported into R from external sources, like a csv or txt file.
- We’ll import data from csv files using the **read_csv()** function from **tidyverse**.
 - **read_csv()** reads csv files (values separated by “,” or “;”).
 - **read_delim()** reads files in which values are separated by any delimiter.

- These functions have many arguments that make them very flexible and allow users to import basically any kind of table stored in a text file. Check out `?read_delim`.
- In base R, the corresponding functions are `read.csv()` and `read.table()`.
- Data can also be imported into R from most external file formats (SAS, SPSS, Stata, Excel, etc.) using the tidyverse packages `readxl` and `haven`, or the `foreign` package in traditional R.
- Note that you can click on a data frame's name in RStudio's Environment pane. That will open the data frame in a window, similar to SPSS's data view.

```
# Normally we create data frames by importing data from external files, for
# example csv files.
data <- read_csv("./data/data.csv")

# View the result.
data

## # A tibble: 1,000 x 5
##   1. Il tuo anno di nascita è...~1 2. Il tuo genere è...~2 4. Sei nato/a in Ita~3
##   <dbl> <chr> <chr>
## 1 1994 Uomo Si
## 2 1968 Donna No
## 3 1969 Uomo Si
## 4 1964 Uomo Si
## 5 1971 Uomo Si
## 6 1961 Preferisco non rispon~ Si
## 7 1993 Donna Si
## 8 1981 Uomo Si
## 9 1979 Donna Si
## 10 1971 Uomo Si
## # i 990 more rows
## # i abbreviated names: 1: `1. Il tuo anno di nascita è...`,
## # 2: `2. Il tuo genere è...`, 3: `4. Sei nato/a in Italia?`
## # i 2 more variables:
## # `11. Il tuo ultimo titolo di studi conseguito è...` <chr>, company <chr>
# Note the different pieces of information that are displayed when printing a tibble
# data frame.
```

2.2.5 Lists

- A list is simply a **collection** of objects. It can contain any kind of object, with no restriction. A list can contain other lists.
- Lists have `list` as *type and class*.
- Data frames are a type of list. Other complex objects in R are also *stored*

as lists (have `list` as type), although their `class` is not `list`: for example, results from statistical estimations or network community detection procedures.

- Use `str(list)` to display the types and lengths of elements in a list.
- List may be *named*, that is, have element names. You can view or assign names with the `names` function (base R) or the `set_names` function (tidyverse).
- **Three different notations to index lists:**
 1. `[]` notation, e.g. `my.list[3]`.
 2. `[[]]` notation, e.g. `my.list[[3]]` or `my.list[["element.name"]]`.
 3. The `$` notation. This only works for named lists. E.g., `list$element.name`. This is the same as the `[[]]` notation: `list$element.name` is the same as `list[["element.name"]]` or `list[[i]]` (where `i` is the position of the element called *element.name* in the list).
- These three indexing methods work in exactly the same way as for data frames (see Section 2.4.1).
- **What we do in the following code.**
 - Create, display, and index a list.

```
# Let's get some objects to put in a list.

# A simple numeric vector.
(num <- 1:10)

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

# A matrix.
(mat <- matrix(1:4, nrow=2, ncol=2))

##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4

# A character vector.
(char <- colors()[1:5])

## [1] "white"      "aliceblue"    "antiquewhite" "antiquewhite1"
## [5] "antiquewhite2"

# Create a list that contains all these objects.
L <- list(num, mat, char)

# Display it
L

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

```
## [[2]]
##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4
##
## [[3]]
## [1] "white"          "aliceblue"          "antiquewhite" "antiquewhite1"
## [5] "antiquewhite2"

# Create a named list
L <- list(numbers= num, matrix= mat, colors= char)

# Display it
L

## $numbers
## [1]  1  2  3  4  5  6  7  8  9 10
##
## $matrix
##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4
##
## $colors
## [1] "white"          "aliceblue"          "antiquewhite" "antiquewhite1"
## [5] "antiquewhite2"

# Type and class
typeof(L)

## [1] "list"

class(L)

## [1] "list"

# Extract the first element of L
# [ ] notation (result is a list containing the element).
L[1]

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

# [[ ]] notation (result is the element itself, no longer in a list).
L[[1]]

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

# $ notation (result is the element itself, no longer in a list).
L$numbers
```

```
## [1] 1 2 3 4 5 6 7 8 9 10
# Name indexing.
L[["numbers"]]

## [1] 1 2 3 4 5 6 7 8 9 10
# Types of elements in L.
str(L)

## List of 3
## $ numbers: int [1:10] 1 2 3 4 5 6 7 8 9 10
## $ matrix : int [1:2, 1:2] 1 2 3 4
## $ colors : chr [1:5] "white" "aliceblue" "antiquewhite" "antiquewhite1" ...
```

2.3 Arithmetic, statistical, and relational operations

2.3.1 Arithmetic operations and recycling

- R can work as a normal calculator.
 - Addition/subtraction: $7+3$
 - Multiplication: $7*3$
 - Negative: -7
 - Division: $7/3$
 - Integer division: $7\%/\%3$
 - Integer remainder: $7\%\%3$
 - Exponentiation: 7^3
- Many operations involving vectors in R are performed **element-wise**, i.e., separately on each element of the vector (see examples below).
- Most operations on vectors use the **recycling** rule: if a vector is too short, its values are re-used the number of times needed to match the desired length (see examples below).
- Examples of vector operations and recycling:
 - $[1\ 2\ 3\ 4] + [1\ 2\ 3\ 4] = [1+1\ 2+2\ 3+3\ 4+4]$ (element-wise addition.)
 - $[1\ 2\ 3\ 4] + 1 = [1+1\ 2+1\ 3+1\ 4+1]$ (1 is recycled 3 times to match the `length` of the first vector.)
 - $[1\ 2\ 3\ 4] + [1\ 2] = [1+1\ 2+2\ 3+1\ 4+2]$ ($[1\ 2]$ is recycled once.)
 - $[1\ 2\ 3\ 4] + [1\ 2\ 3] = [1+1\ 2+2\ 3+3\ 4+1]$ ($[1\ 2\ 3]$ is recycled one third of a time: R will warn that the length of longer vector is not a multiple of the length of shorter vector.)

2.3.2 Relational operations and logical vectors

- **Relational** operators: `>`, `<`, `<=`, `>=`. Equal is `==` (NOT `=`). *Not* equal is `!=`.
 - Note: equal is `==`, whereas `=` has a different meaning. `=` is used to assign function arguments (e.g. `matrix(x, nrow = 3, ncol = 4)`), or to assign objects (`x <- 2` is the same as `x = 2`).
- Relational operations result in **logical** vectors: vectors of TRUE/FALSE values.
- Like arithmetic operations, relational ones are performed element-wise on vectors, and recycling applies.
- Logical operators: `&` for AND, `|` for OR.
- Negation (i.e. opposite) of a logical vector: `!`.
- Is value x in vector y ? `x %in% y`.
- R can convert logical vectors to numeric (`as.numeric()`, `as.integer()`). In this conversion, TRUE becomes 1 and FALSE becomes 0. Conversely, if converted to logical (`as.logical()`), 1/0 are TRUE/FALSE.
 - Therefore, if `x` is a logical vector, `sum(x)` gives the *count* of TRUES in `x` (sum of 1s in the vector).
 - `mean(x)` gives the *proportion* of 1s in `x` (mean of a binary vector: sum of 1s in the vector divided by number of elements in the vector).

2.3.3 Examples of arithmetic and statistical functions

- Arithmetic *vector* functions are performed element-wise, and return a vector. Examples:
 - Exponential: `exp(x)`.
 - Logarithm: `log(x)` (base e) or `log10(x)` (base 10).
- Statistical *scalar* functions are executed on the set of all vector elements taken together, and return a scalar. Examples:
 - `mean(x)` and `median(x)`.
 - Standard deviation and variance: `sd(x)`, `var(x)`.
 - Minimum and maximum: `min(x)`, `max(x)`.
 - `sum(x)`: sum of all elements in `x`.
- `table(x)` is another basic statistical function (but it's not scalar):
 - `table(x)` returns a **table** object with the absolute frequencies of values in `x`.
- Functions such as `sum()`, `mean()` and `table()` are very useful when programming in R (for example, when writing your own functions). However, if you just need descriptive statistics for the data, there are more convenient tools you can use. Some of these are the following functions, which work well with **tidyverse**:
 - The `skim` function (from the `skimr` package) for descriptive statistics of continuous/quantitative variables.
 - The `tabyl` function (from the `janitor` package) for frequencies of categorical variables.

2.3.4 Missing and infinite values

- Missing values in R are represented by `NA` (Not Available).
 - If your data has a different code for missing values (e.g., -99), you'll have to recode that to `NA` for R to properly handle missing values in your data.
- Infinity may result from arithmetic operations: `Inf` and `-Inf` (e.g. `3/0`). `NaN` also may result, meaning Not a Number (e.g. `0/0`).
 - While `NA`s can appear in any type of object, `Inf`, `-Inf` and `NaN` can only appear in numeric objects.
- `is.na(x)` checks if each element of `x` is `NA` and returns `TRUE` if that's the case, `FALSE` otherwise. It's a vector function (its value has the same `length` as `x`).

*# Just a few arithmetic operations between vectors to demonstrate element-wise
calculations and the recycling rule.*

```
(v1 <- 1:4)
```

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

```
(v2 <- 1:4)
```

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

```
# [1 2 3 4] + [1 2 3 4]  
v1 + v2
```

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

```
# [1 2 3 4] + 1  
1:4 + 1
```

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

```
# [1 2 3 4] + [1 2]  
(v1 <- 1:4)
```

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

```
(v2 <- 1:2)
```

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

```
v1 + v2
```

```
## [1] 2 4 4 6
```

```
# [1 2 3 4] + [1 2 3]  
1:4 + 1:3
```

```
## Warning in 1:4 + 1:3: longer object length is not a multiple of shorter object  
## length
```

2.3. ARITHMETIC, STATISTICAL, AND RELATIONAL OPERATIONS 23

```
## [1] 2 4 6 5
```

```
# Let's do a simple arithmetic operation to get respondents' age in years.
```

```
# First, let's take a single variable from the ego attribute data: ego's year of  
# birth for the first 10 respondents. Let's put the result in a separate vector.  
# This code involves indexing, we'll explain that better below.
```

```
# Respondents' year of birth  
year <- data[[1]]
```

```
# Let's see the first 10 values (i.e., first 10 respondents)  
year[1:10]
```

```
## [1] 1994 1968 1969 1964 1971 1961 1993 1981 1979 1971
```

```
# Age for the first 10 respondents  
(age <- 2023-year[1:10])
```

```
## [1] 29 55 54 59 52 62 30 42 44 52
```

```
# Relational operations.
```

```
# Note how the following comparisons are performed element-wise, and the value  
# to which age is compared (30) is recycled.
```

```
# Is age equal to 30?  
age==30
```

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

```
# The resulting logical vector is TRUE for those elements (i.e., respondents)  
# who meet the condition.
```

```
# Which respondent's age is greater than 40?  
age > 40
```

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

```
# Which respondent age values are lower than 40 OR greater than 60?  
age < 40 | age > 60
```

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

```
# Which elements of "age" are lower than 40 AND greater than 30?  
age < 40 & age > 30
```

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

```
# Notice the difference between OR (/) and AND (&).
```

```
# Is 30 in "age"? I.e., is one of the respondents of 30 years of age?
30 %in% age
```

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

```
# Is "age" in c(30, 42)? That is, which values of "age" are either 30 or 35?
age %in% c(30, 42)
```

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

```
# A logical vector can be converted to numeric: TRUE becomes 1 and FALSE becomes
# 0.
```

```
age > 45
```

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

```
as.numeric(age > 45)
```

```
## [1] 0 1 1 1 1 1 0 0 0 1
```

```
# This allows us to use the sum() and the mean() functions to get the count and
# proportion of TRUE's in a logical vector.
```

```
# Count of TRUE's: Number of respondents (elements of "age") that are older than
# 40.
```

```
sum(age > 45)
```

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

```
# How many respondents in the vector are older than 30?
```

```
sum(age > 30)
```

```
## [1] 8
```

```
# Proportion of TRUE's: What's the proportion of "age" elements (respondents)
# that are greater than 50?
```

```
mean(age > 50)
```

```
## [1] 0.6
```

```
# ***** EXERCISES
```

```
#
```

```
# (1) Obtain a logical vector indicating which elements of "age" are smaller than 30
# OR greater than 50. Then obtain a logical vector indicating which elements of
# "age" are smaller than 30 AND greater than 50. Why all elements are FALSE in
# the latter vector?
```

```
#
```

```
# (2) Using the : shortcut, create a vector that goes from 1 to 100 in steps of 1.
# Obtain a logical vector that is TRUE for the first 10 elements and the last 10
# elements of the vector.
```



```
#
# (3) Use the age vector with relational operators and sum/mean to answer these
# questions: How many respondents are younger than 50? What percentage of
# respondents is between 30 and 40 years of age, including 30 and 40? Is there
# any respondent who is younger than 20 OR older than 70 (use any())?
#
# (4) How many elements of the vector 1:100 are greater than the length of that
# vector divided by 2? Use sum().
#
# *****
```

2.4 Subsetting

- **Subsetting** is crucial in R. It means extracting one (or more) of an object's elements or components, typically by appending an index (or subscript) to that object (this is also called “indexing” or “subscripting”).
- Subsetting can be used to **extract** (view, query) the component of an object, or to **replace** it (assign a different value to that element).
- The basic notation for subsetting in R is `[]`: `x[i]` gives you the i -th element of object `x`.
- **Numeric subsetting** uses integers in square brackets `[]`: e.g. `x[3]`. Note that you can use negative integers to index (select) everything *but* that element: e.g. `x[-3]`, `x[c(-2,-4)]`.
- **Logical subsetting** uses logical vectors in square brackets `[]`. It's used to subset objects based on a condition, e.g., to index all values in `x` that are greater than 3 (see example code below).
- **Name subsetting** uses element names. Elements in a vector, and rows or columns in a matrix can have names.
 - Names can be displayed and assigned using the `names` function in base R, or `set_names` (just to assign names) in tidyverse.
- When subsetting you must take into account the **number of dimensions** of an object. For example, vectors have one dimension, matrices have two. Arrays can be defined with three dimensions or more (e.g. three-way tables).
 - Square brackets typically contain a slot for each dimension of the object, separated by a comma:
 - * `x[i]` indexes the i -th element of the one-dimensional object `x`;
 - * `x[i,j]` indexes the i,j -th element of the two-dimensional object `x` (e.g. `x` is a matrix, i refers to a row and j refers to a column);
 - * `x[i,j,k]` indexes the i,j,k -th element of the three-dimensional object `x`, etc.
 - Notice that a dimension's slot may be empty, meaning that we index all elements in that dimension. So, if `x` is a matrix, `x[3,]` will index the whole 3rd row of the matrix – i.e. [row 3, all columns].

- If `x` has more than one dimension (e.g. it's a matrix), then `x[3]` (no comma, just one slot) is still valid, but it might give you unexpected results.
- Matrices have special functions that can be used for subsetting, e.g. `diagonal()`, `upper.tri()`, `lower.tri()`. These can be useful for manipulating adjacency matrices.
- Particular subsetting rules may apply to particular classes of objects, for example lists and data frames (see next Section 2.4.1).

2.4.1 Subsetting data frames

- **List notations.** Data frames are a special class of lists. Just like any list, data frames can be subset in the following three ways:
 1. `[]` notation, e.g. `df[3]` or `df["variable.name"]`. This returns another data frame that only includes the indexed element(s), e.g. only the 3rd element. Note:
 - This notation *preserves the data.frame class*: the result is still a data frame.
 - This notation can be used to index *multiple* elements of a data frame into a new data frame, e.g. `df[c(1,3,5)]` or `df[c("sex", "age")]`
 2. `[[]]` notation, e.g. `df[[3]]` or `df[["variable.name"]]`. This returns the specific element (column) selected, not as a data frame but as a vector with its own type and class, e.g. the numeric vector *within* the 3rd element of `df`. Note two differences from the `[]` notation:
 - `[[]]` *does not* preserve the `data.frame` class. The result is *not* a data frame.
 - Consistently, `[[]]` can only be used to index a *single* element (column) of the data frame, not multiple elements.
 3. The `$` notation. If *variable.name* is the name of a specific variable (column) in `df`, then `df$variable.name` indexes that variable. This is the same as the `[[]]` notation: `df$variable.name` is the same as `df[["variable.name"]]`, and it's also the same as `df[[i]]` (where `i` is the position of the variable called *variable.name* in the data frame).
- **Matrix notation.** Data frames can also be subset like a matrix, with the `[,]` notation:
 - `df[2,3]`, `df[2,]`, `df[,3]`.
 - `df[, "age"]`, `df[, c("sex", "age")]`, `df[5, "age"]`
- **Keep in mind the difference** between the following:
 - Extracting a data frame's variable (column) in itself, as a vector (numeric, character, etc.) — The single pepper packet by itself in the figure below (panel C). This is given by `df[[i]]`, `df[["variable.name"]]`, `df$variable.name`.
 - Extracting another data frame of just one variable (column) — The single pepper packet *within* the pepper shaker in the figure below (panel B). This is given by `df[i]`, `df["variable.name"]`.

- **Subsetting verbs** in tidyverse. In addition to subsetting data frames via the base indexing syntax described above, we can also use the subsetting functions introduced by the `dplyr` package in tidyverse (see the next chapter).



```
# Numeric subsetting
```

```
# -----
```

```
# Let's use our vector of ego age values again.
```

```
age
```

```
## [1] 29 55 54 59 52 62 30 42 44 52
```

```
# Index its 2nd element: The age of the 2nd respondent.
```

```
age[2]
```

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

```
# Its 2nd, 4th and 5th elements.
```

```
age[c(2,4,5)]
```

```
## [1] 55 59 52
```

```
# Fifth to seventh elements
```

```
age[5:7]
```

```
## [1] 52 62 30
```

```
# Use indexing to assign (replace) an element.
```

```
age[2] <- 45
```

```
# The content of x has now changed.
```

```
age
```

```
## [1] 29 45 54 59 52 62 30 42 44 52
```

```
# Let's subset the adjacency matrix we created before.
adj
```

```
##      [,1] [,2] [,3]
## [1,]    0    1    1
## [2,]    1    0    1
## [3,]    0    0    0
```

```
# Its 2,3 cell: Edge from node 2 to node 3.
adj[2,3]
```

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

```
# Its 2nd column: All edges to node 2.
adj[,2]
```

```
## [1] 1 0 0
```

```
# Its 2nd and 3rd row: All edges from nodes 2 and 3.
adj[2:3,]
```

```
##      [,1] [,2] [,3]
## [1,]    1    0    1
## [2,]    0    0    0
```

```
# Logical subsetting
```

```
# - - - - -
```

```
# Which values of "age" are between 40 and 60?
```

```
# Let's create a logical index that flags these values.
(ind <- age > 40 & age < 60)
```

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

```
# Use this index to extract these values from vector "age" via logical subsetting.
age[ind]
```

```
## [1] 45 54 59 52 42 44 52
```

```
# We could also have typed directly:
age[age > 40 & age < 60]
```

```
## [1] 45 54 59 52 42 44 52
```

```
# Subsetting data frames
```

```
# - - - - -
```

```
# We'll use our data frame (just its first 20 rows).
data.20 <- slice(data, 1:20)
```

```
# Numeric subsetting works on data frames too: it allows you to index variables.
```

```
# The 3rd variable.
```

```
data.20[3]
```

```
## # A tibble: 20 x 1
##   `4. Sei nato/a in Italia?`
##   <chr>
## 1 Sì
## 2 No
## 3 Sì
## 4 Sì
## 5 Sì
## 6 Sì
## 7 Sì
## 8 Sì
## 9 Sì
## 10 Sì
## 11 Sì
## 12 Sì
## 13 Sì
## 14 Sì
## 15 Sì
## 16 Sì
## 17 Sì
## 18 Sì
## 19 Sì
## 20 Sì
```

```
# Note the difference with the double square bracket.
```

```
data.20[[3]]
```

```
## [1] "Sì" "No" "Sì" "Sì" "Sì" "Sì" "Sì" "Sì" "Sì" "Sì" "Sì" "Sì" "Sì" "Sì"
## [16] "Sì" "Sì" "Sì" "Sì" "Sì"
```

```
# What do you think is the difference?
```

```
class(data.20[3])
```

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

```
class(data.20[[3]])
```

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

```
# The [[ ]] notation extracts the actual column as a vector, while [ ] keeps
# the data frame class.
```

```
# We can also subset data frames as matrices.
# The second and third columns.
```

```
data.20[,2:3]
```

```
## # A tibble: 20 x 2
##   `2. Il tuo genere è...` 4. Sei nato/a in Ita~1
##   <chr> <chr>
## 1 Uomo Sì
## 2 Donna No
## 3 Uomo Sì
## 4 Uomo Sì
## 5 Uomo Sì
## 6 Preferisco non rispondere Sì
## 7 Donna Sì
## 8 Uomo Sì
## 9 Donna Sì
## 10 Uomo Sì
## 11 Uomo Sì
## 12 Donna Sì
## 13 Donna Sì
## 14 Altra identità di genere (transgender, non binario, e~ Sì
## 15 Uomo Sì
## 16 Donna Sì
## 17 Donna Sì
## 18 Uomo Sì
## 19 Uomo Sì
## 20 Uomo Sì
## # i abbreviated name: 1: `4. Sei nato/a in Italia?`
```

```
# Lines 1 to 3
```

```
data.20[1:3,]
```

```
## # A tibble: 3 x 5
##   `1. Il tuo anno di nascita è...` 2. Il tuo genere è...~1 4. Sei nato/a in Ita~2
##   <dbl> <chr> <chr>
## 1 1994 Uomo Sì
## 2 1968 Donna No
## 3 1969 Uomo Sì
## # i abbreviated names: 1: `2. Il tuo genere è...`,
## # 2: `4. Sei nato/a in Italia?`
## # i 2 more variables:
## # `11. Il tuo ultimo titolo di studi conseguito è...` <chr>, company <chr>
```

```
# We can use name indexing with data frames, selecting variables by name
```

```
data.20["company"]
```

```
## # A tibble: 20 x 1
##   company
##   <chr>
```

```
## 1 My Vegetarian Dinner
## 2 Urban Gallery
## 3 Office Tile
## 4 Raven
## 5 Fix Guru
## 6 Office Brush
## 7 House Brush
## 8 Satan's Sister
## 9 FruityFlix
## 10 Coal Kings
## 11 Coal Kings
## 12 Coal Kings
## 13 Coal Kings
## 14 Beep Sports
## 15 The Auto DNA
## 16 Office Tile
## 17 Coal Kings
## 18 Wood Works
## 19 Bloom Marketing
## 20 Coal Kings
```

```
data.20[["company"]]
```

```
## [1] "My Vegetarian Dinner" "Urban Gallery"      "Office Tile"
## [4] "Raven"                "Fix Guru"           "Office Brush"
## [7] "House Brush"          "Satan's Sister"     "FruityFlix"
## [10] "Coal Kings"           "Coal Kings"         "Coal Kings"
## [13] "Coal Kings"           "Beep Sports"        "The Auto DNA"
## [16] "Office Tile"          "Coal Kings"         "Wood Works"
## [19] "Bloom Marketing"      "Coal Kings"
```

The \$ notation is very common and concise. It's equivalent to the [[notation.

```
data.20$company
```

```
## [1] "My Vegetarian Dinner" "Urban Gallery"      "Office Tile"
## [4] "Raven"                "Fix Guru"           "Office Brush"
## [7] "House Brush"          "Satan's Sister"     "FruityFlix"
## [10] "Coal Kings"           "Coal Kings"         "Coal Kings"
## [13] "Coal Kings"           "Beep Sports"        "The Auto DNA"
## [16] "Office Tile"          "Coal Kings"         "Wood Works"
## [19] "Bloom Marketing"      "Coal Kings"
```

This is the same as data.20[[3]] or data.20[["ego.age"]]

```
identical(data.20[[5]], data.20$company)
```

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

```

# With tidyverse, this type of subsetting syntax is replaced by new "verbs"
# (see next chapter):
# * Index data frame rows: filter() instead of []
# * Index data frame columns: select() instead of []
# * Extract data frame variable as a vector: pull() instead of [[]] or $

# ***** EXERCISES
#
# Create the fictitious variable var <- c(1:30, rep(NA, 3), 34:50). Use is.na()
# to index all the NA values in the variable. Then use is.na() to index all
# values that are not NA. Hint: Remember the operator used to negate a logical
# vector. Finally, use this indexing to remove all NA values from var.
#
# *****

```

2.5 Pipes and the `|>` operator

- The original pipe operator, `%>%`, was introduced by the `magrittr` package in 2014. It quickly gained popularity in the R community and was adopted by `tidyverse` (and other packages). In 2021, R incorporated the pipe idea with a new, similar (but not identical) operator: `|>`. See this page for an overview of the differences between `|>` and `%>%`.
- The idea behind pipes is in essence very simple:
 - `f(g(x))` becomes `x |> g() |> f()`.
 - For example: `mean(table(x))` becomes `x |> table() |> mean()`.
- So `|>` pipes the output of the previous function (e.g., `table()`) into the input of the following function (e.g., `mean()`). This turns inside-to-outside code into left-to-right code. Because left to right is the direction most of us are used to read in (at least in English and other Western languages), pipes make R code easier to read and follow.
- You may also see pipes concatenating multiple lines of code. That's possible and a common coding style. Instead of

```
x |> table() |> mean()
```

you can write

```
x |>
  table() |>
  mean()
```

Let's see how this works with some of our data objects.

```

# Respondent gender variable
gender <- data[[2]]

```



```

# First 10 values
gender[1:10]

## [1] "Uomo" "Donna"
## [3] "Uomo" "Uomo"
## [5] "Uomo" "Preferisco non rispondere"
## [7] "Donna" "Uomo"
## [9] "Donna" "Uomo"

# Frequency of gender categories in the data
table(gender)

## gender
## Altra identità di genere (transgender, non binario, ecc.)
## 3
## Donna
## 522
## Preferisco non rispondere
## 8
## Uomo
## 467

# Average frequency
mean(table(gender))

## [1] 250

# Let's re-write this with the pipe operator
gender |>
  table() |>
  mean()

## [1] 250

```

2.6 Writing your own R functions

- One of the most powerful tools in R is the ability to **write your own functions**.
- A function is a **piece of code** that operates on one or multiple **arguments** (the *input*), and returns an *output* (the function **value** in R terminology). Everything that happens in R is done by a function.
- Many R functions have **default values** for their arguments: if you don't specify the argument's value, the function will use the default.
- Once you write a function and define its arguments, you can run that function on any argument values you want — provided that the function code actually works on those argument values.
- R functions, combined with functionals and summarization methods, are

the best way to run exactly the same code on many different objects. Functions are **crucial for code reproducibility** in R. If you write functions, you won't need to re-write (copy and paste) the same code over and over again — you just write it once in the function, then run the function any time and on any arguments you need. This yields clearer, shorter, more readable code with less errors.

- New functions are also commonly used to **redefine existing functions** by pre-setting the value of specific arguments. For example, if you want all your plots to have **red** as color, you can take R's existing plotting function `plot()`, and wrap it in a new function that always executes `plot()` with the argument `col="red"`. Your function would be something like `my.plot <- function(...) {plot(..., col="red")}`.
- **Tips and tricks** with functions:
 - `stopifnot()` is useful to check that function arguments are of the type that was intended by the function author. It stops the function if a certain condition is not met by a function argument (e.g., argument is *not* a **numeric** object, if the function was written for **numeric** objects).
 - `return()` allows you to explicitly set the output that the function will return (clearer code). It is also used to stop function execution earlier under certain conditions. Note: If you don't use `return()`, the function value (output) is the last object that is printed at the end of the function code.
 - `if` is a flow control tool that is frequently used within functions: it specifies what the function should do **if** a certain condition is met at one point.
 - First think particular, then generalize. When you want to write a function, it's a good idea to first try the code on a “real”, specific existing object in your workspace. If the code does what you want on that object, you can then wrap it into a general function to be run on any similar object (see examples in the code below).

Any piece of code you can write and run in R, you can also put in a function.

Let's write a trivial function that takes its argument and multiplies it by 2.

```
times2 <- function(x) {
  x*2
}
```

Now we can run the function on any argument.

```
times2(x= 3)
```

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

```
times2(x= 10)
```

```
## [1] 20
```

```
times2(50)
```

```
## [1] 100
```

```
# A function that takes its argument and prints a sentence with it:  
myoutput <- function(word) {  
  print(paste("My output is", word))  
}
```

```
# Let's run the function.  
myoutput("cat")
```

```
## [1] "My output is cat"
```

```
myoutput(word= "table")
```

```
## [1] "My output is table"
```

```
myoutput("any word here")
```

```
## [1] "My output is any word here"
```

```
# Not a particularly useful function...
```

```
# Note that the function output is the last object that is printed at the end  
# of the function code.
```

```
times2 <- function(x) {  
  y <- x*2  
  y  
}  
times2(x=4)
```

```
## [1] 8
```

```
# If nothing is printed, then the function returns nothing.
```

```
times2 <- function(x) {  
  y <- x*2  
}  
times2(x=4)
```

```
# A function will return an error if it's executed on arguments that are not  
# suitable for the code inside the function. E.g., R can't multiply "a" by 2...
```

```
times2 <- function(x) {  
  x*2  
}  
times2(x= "a")
```

```
## Error in x * 2: non-numeric argument to binary operator
```

```
# Let's then specify that the function's argument must be numeric.
times2 <- function(x) {
  stopifnot(is.numeric(x))
  x*2
}

# Let's try it now.
times2(x= "a")
```

```
## Error in times2(x = "a"): is.numeric(x) is not TRUE
```

```
# This still throws an error, but it makes the error clearer to the user and
# it immediately indicates where the problem is.
```

```
# Using if, we can also re-write the function so that it returns NA with a
# warning if its argument is not numeric -- instead of just stopping with an
# error.
```

```
times2 <- function(x) {
  # If x is not numeric
  if(!is.numeric(x)) {
    # Give the warning
    warning("Your argument is not numeric!", call. = FALSE)
    # Return missing value
    return(NA)
    # Otherwise, return x*2
  } else {
    return(x*2)
  }
}

# Try the function
times2(2)
```

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

```
times2("a")
```

```
## Warning: Your argument is not numeric!
```

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

2.7 Types and classes of objects

This is a quick summary of the basics about **types** and **classes** of objects in R.

- Three functions are used to know what kind of object you are dealing with in R: `class()`, `mode()`, and `typeof()`.

- For most purposes, you only need to know what the **class** of an object is. This is returned by `class()`. The class of an object determines what R functions you can or cannot run on that object, and how functions will behave when you run them on the object. In particular, if the function has a **method** for a specific class *A* of objects, it will use that method whenever an object of class *A* is given as its argument.
- `typeof()` and `mode()` return the **type** and **mode** of an object, respectively. Although they refer to slightly different classifications of objects, type and mode give essentially the same kind of information — the type of data structure in which the object is stored, also called the R “internal type” or “storage mode”. For example, an object can be internally stored in R as double-precision numbers, integer numbers, or character strings.
 - You should prefer `typeof()` over `mode()`. `mode()` refers to the old S classification of types and is mostly used for S compatibility.
- While most times all you need to know is the **class** of an object, there are a few cases in which knowing the **type** is useful too. For example, you may want to know the **type** of a matrix object (whose **class** is always **matrix**) to check if the values in the matrix are being stored as numbers or character strings (that will affect the result of some functions).
- Main classes/types of objects
 - **numeric**: Numerical data (integer, real or complex numbers).
 - **logical**: TRUE/FALSE data.
 - **character**: String data.
 - **factor**: Categorical data, that is, integer numbers with string labels attached. May be *unordered* factors (nominal data) or *ordered* factors (ordinal data).
- Special and complex classes/types
 - **list**: A collection of elements of any type, including numeric, character, logical.
 - **data.frame**: A dataset. In R, a data frame is a special kind of list (its type is **list** but its class is **data.frame**), where each variable (column) is a list element (see Section 2.2.4)
 - **matrix**: Matrix values can be numeric, character, logical etc. So an object can have **matrix** as class and **numeric**, **character** or **logical** as type. While data frames can contain variables of different type (e.g. a character variable and a numeric variable), matrices can only contain values of *one* type.
 - Functions (more on this in Section 2.6).
 - Expressions.
 - Formulas.
 - Other objects: Statistical results (e.g. linear model estimates), dendrograms, graphics objects, etc.

- Relevant functions

- `class()`, `typeof()` and `mode()`, as discussed above.
- *is.type* functions verify that an object is in a specific type or class: e.g. `is.numeric(x)`, `is.character(x)` (they return TRUE or FALSE).
- *as.type* functions convert objects between types or classes: e.g. `as.numeric()`, `as.character()`. If the conversion is impossible, the result is NA: e.g. `as.numeric("abc")` returns NA.

```
# A numeric vector of integers.
```

```
n <- 1:100
```

```
# Let's check the class and type.
```

```
class(n)
```

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

```
typeof(n)
```

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

```
# A character object.
```

```
(char <- c("a", "b", "c", "d", "e", "f"))
```

```
## [1] "a" "b" "c" "d" "e" "f"
```

```
# Class and type.
```

```
class(char)
```

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

```
typeof(char)
```

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

```
# Let's put n in a matrix.
```

```
(M <- matrix(n, nrow=10, ncol=10))
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
## [1,]  1   11   21   31   41   51   61   71   81   91
## [2,]  2   12   22   32   42   52   62   72   82   92
## [3,]  3   13   23   33   43   53   63   73   83   93
## [4,]  4   14   24   34   44   54   64   74   84   94
## [5,]  5   15   25   35   45   55   65   75   85   95
## [6,]  6   16   26   36   46   56   66   76   86   96
## [7,]  7   17   27   37   47   57   67   77   87   97
## [8,]  8   18   28   38   48   58   68   78   88   98
## [9,]  9   19   29   39   49   59   69   79   89   99
## [10,] 10   20   30   40   50   60   70   80   90  100
```

```

# Class/type of this object.
class(M)

## [1] "matrix" "array"
# Type and mode tell us that this is an *integer* matrix.
typeof(M)

## [1] "integer"
# There are character and logical matrices too.
char

## [1] "a" "b" "c" "d" "e" "f"
(C <- matrix(char, nrow=3, ncol= 2))

##      [,1] [,2]
## [1,] "a"  "d"
## [2,] "b"  "e"
## [3,] "c"  "f"
# Class and type.
class(C)

## [1] "matrix" "array"
typeof(C)

## [1] "character"
# Notice that a matrix can contain numbers but still be stored as character.
(M <- matrix(c("1", "2", "3", "4"), nrow=2, ncol=2))

##      [,1] [,2]
## [1,] "1"  "3"
## [2,] "2"  "4"
class(M)

## [1] "matrix" "array"
typeof(M)

## [1] "character"
# Let's convert "char" to factor.
char

## [1] "a" "b" "c" "d" "e" "f"
(char <- as.factor(char))

## [1] a b c d e f

```

```
## Levels: a b c d e f
# This means that now char is not just a collection of strings, it is a
# categorical variable in R's mind: it is a collection of numbers with character
# labels attached.

# Compare the different behavior of as.numeric(): char as character...
(char <- c("a", "b", "c", "d", "e", "f"))

## [1] "a" "b" "c" "d" "e" "f"

# Convert to numeric
as.numeric(char)

## Warning: NAs introduced by coercion
## [1] NA NA NA NA NA NA

# ...versus char as factor.
char <- c("a", "b", "c", "d", "e", "f")
(char <- as.factor(char))

## [1] a b c d e f
## Levels: a b c d e f
as.numeric(char)

## [1] 1 2 3 4 5 6
# char is a different object in R's mind when it's character vs when
# it's factor. Characters can't be converted to numbers,
# but factors can.
```


Chapter 3

Data wrangling and descriptive statistics

This workshop session is about data wrangling and obtaining summary or descriptive statistics about the data. We will mostly (but not exclusively) use tidyverse packages and functions.

A lot of what's covered here is treated in more details in the `dplyr` package website. Also see the `dplyr` cheat sheet for a visually nice summary.

For more information about the many different types of external sources from which (rectangular) data can be imported into R, see the `readr` package website.

Let's start by loading packages, clearing the workspace, and loading the data from external source.

```
# Load packages
library(tidyverse)
library(skimr)
library(janitor)

# Clear the workspace
rm(list = ls())

# Let's read the data in as a tibble data frame.
(data <- read_csv("../data/data.csv"))
```

```
## # A tibble: 1,000 x 5
##   1. Il tuo anno di nascita è...~1 2. Il tuo genere è...~2 4. Sei nato/a in Ita~3
##                                <dbl> <chr>                                <chr>
## 1                                1994 Uomo                                Sì
## 2                                1968 Donna                                No
```

```
## 3      1969 Uomo      Si
## 4      1964 Uomo      Si
## 5      1971 Uomo      Si
## 6      1961 Preferisco non rispon~ Si
## 7      1993 Donna      Si
## 8      1981 Uomo      Si
## 9      1979 Donna      Si
## 10     1971 Uomo      Si
## # i 990 more rows
## # i abbreviated names: 1: `1. Il tuo anno di nascita è...`,
## # 2: `2. Il tuo genere è...`, 3: `4. Sei nato/a in Italia?`
## # i 2 more variables:
## # `11. Il tuo ultimo titolo di studi conseguito è...` <chr>, company <chr>
```

3.1 Subsetting and renaming variables

- The `select()` verb.
- Renaming variables (i.e., data columns).
- Selecting (i.e., subsetting) variables.
- Pulling one variable out of the data frame.

```
# Give better names to the variables: by variable name.
data |>
  rename(
    birth.y = `1. Il tuo anno di nascita è...`,
    gender = `2. Il tuo genere è...`
  )
```

```
## # A tibble: 1,000 x 5
##   birth.y gender      4. Sei nato/a in Ita~ 11. Il tuo ultimo ti~2 company
##   <dbl> <chr>      <chr>      <chr>      <chr>
## 1  1994 Uomo      Si      Master post-laurea  My Veg~
## 2  1968 Donna     No      Laurea            Urban ~
## 3  1969 Uomo      Si      Diploma di scuola sup~ Office~
## 4  1964 Uomo      Si      Diploma di scuola sup~ Raven
## 5  1971 Uomo      Si      Diploma di scuola sup~ Fix Gu~
## 6  1961 Preferisco non~ Si      Laurea            Office~
## 7  1993 Donna      Si      Scuola di specializza~ House ~
## 8  1981 Uomo      Si      Laurea            Satan'~
## 9  1979 Donna      Si      Laurea            Fruity~
## 10 1971 Uomo      Si      Diploma di scuola sup~ Coal K~
## # i 990 more rows
## # i abbreviated names: 1: `4. Sei nato/a in Italia?`,
## # 2: `11. Il tuo ultimo titolo di studi conseguito è...`
```

```
# Give better names to the variables: by variable position.
```

```
data |>
  rename(
    birth.y = 1,
    gender = 2
  )
```

```
## # A tibble: 1,000 x 5
##   birth.y gender      4. Sei nato/a in Ita~1 11. Il tuo ultimo ti~2 company
##   <dbl> <chr>      <chr>      <chr>      <chr>
## 1  1994 Uomo      Sì          Master post-laurea  My Veg~
## 2  1968 Donna    No          Laurea            Urban ~
## 3  1969 Uomo      Sì          Diploma di scuola sup~ Office~
## 4  1964 Uomo      Sì          Diploma di scuola sup~ Raven
## 5  1971 Uomo      Sì          Diploma di scuola sup~ Fix Gu~
## 6  1961 Preferisco non~ Sì          Laurea            Office~
## 7  1993 Donna    Sì          Scuola di specializza~ House ~
## 8  1981 Uomo      Sì          Laurea            Satan'~
## 9  1979 Donna    Sì          Laurea            Fruity~
## 10 1971 Uomo      Sì          Diploma di scuola sup~ Coal K~
## # i 990 more rows
## # i abbreviated names: 1: `4. Sei nato/a in Italia?`,
## # 2: `11. Il tuo ultimo titolo di studi conseguito è...`
```

```
# Rename all or a collection of variables with a function: .x represents each
# variable name
```

```
data |>
  rename_with(~ toupper(.x))
```

```
## # A tibble: 1,000 x 5
##   1. IL TUO ANNO DI NASCITA È...~1 2. IL TUO GENERE È...~2 4. SEI NATO/A IN ITA~3
##   <dbl> <chr>      <chr>
## 1  1994 Uomo      Sì
## 2  1968 Donna    No
## 3  1969 Uomo      Sì
## 4  1964 Uomo      Sì
## 5  1971 Uomo      Sì
## 6  1961 Preferisco non rispon~ Sì
## 7  1993 Donna    Sì
## 8  1981 Uomo      Sì
## 9  1979 Donna    Sì
## 10 1971 Uomo      Sì
## # i 990 more rows
## # i abbreviated names: 1: `1. IL TUO ANNO DI NASCITA È...`,
## # 2: `2. IL TUO GENERE È...`, 3: `4. SEI NATO/A IN ITALIA?`
## # i 2 more variables:
```

```
## # `11. IL TUO ULTIMO TITOLO DI STUDI CONSEGUITO È...` <chr>, COMPANY <chr>
```

```
# Final names
```

```
data <- data |>
```

```
  rename(
```

```
    birth.y = `1. Il tuo anno di nascita è...`,
```

```
    gender = `2. Il tuo genere è...`,
```

```
    nato.it = `4. Sei nato/a in Italia?`,
```

```
    edu = `11. Il tuo ultimo titolo di studi conseguito è...`
```

```
  )
```

```
data
```

```
## # A tibble: 1,000 x 5
```

```
##   birth.y gender
```

```
##   <dbl> <chr>
```

```
## 1 1994 Uomo
```

```
## 2 1968 Donna
```

```
## 3 1969 Uomo
```

```
## 4 1964 Uomo
```

```
## 5 1971 Uomo
```

```
## 6 1961 Preferisco non rispondere
```

```
## 7 1993 Donna
```

```
## 8 1981 Uomo
```

```
## 9 1979 Donna
```

```
## 10 1971 Uomo
```

```
## # i 990 more rows
```

```
nato.it edu
```

```
<chr> <chr>
```

```
Si Master post-laurea
```

```
No Laurea
```

```
Si Diploma di scuola superiore
```

```
Si Diploma di scuola superiore
```

```
Si Diploma di scuola superiore
```

```
Si Laurea
```

```
Si Scuola di specializzazione
```

```
Si Laurea
```

```
Si Laurea
```

```
Si Diploma di scuola superiore
```

```
company
```

```
<chr>
```

```
My Veg~
```

```
Urban ~
```

```
Office~
```

```
Raven
```

```
Fix Gu~
```

```
Office~
```

```
House ~
```

```
Satan'~
```

```
Fruity~
```

```
Coal K~
```

```
# Subset to specific variables of interest.
```

```
data |>
```

```
  select(gender, edu)
```

```
## # A tibble: 1,000 x 2
```

```
##   gender
```

```
##   <chr>
```

```
## 1 Uomo
```

```
## 2 Donna
```

```
## 3 Uomo
```

```
## 4 Uomo
```

```
## 5 Uomo
```

```
## 6 Preferisco non rispondere
```

```
## 7 Donna
```

```
## 8 Uomo
```

```
## 9 Donna
```

```
## 10 Uomo
```

```
## # i 990 more rows
```

```
edu
```

```
<chr>
```

```
Master post-laurea
```

```
Laurea
```

```
Diploma di scuola superiore
```

```
Diploma di scuola superiore
```

```
Diploma di scuola superiore
```

```
Laurea
```

```
Scuola di specializzazione post-laurea
```

```
Laurea
```

```
Laurea
```

```
Diploma di scuola superiore
```

```
# Remove certain variables
```

```
data |>
```

```

select(-gender)

## # A tibble: 1,000 x 4
##   birth.y nato.it edu company
##   <dbl> <chr> <chr> <chr>
## 1 1994 Sì Master post-laurea My Vegetarian Dinner
## 2 1968 No Laurea Urban Gallery
## 3 1969 Sì Diploma di scuola superiore Office Tile
## 4 1964 Sì Diploma di scuola superiore Raven
## 5 1971 Sì Diploma di scuola superiore Fix Guru
## 6 1961 Sì Laurea Office Brush
## 7 1993 Sì Scuola di specializzazione post-laurea House Brush
## 8 1981 Sì Laurea Satan's Sister
## 9 1979 Sì Laurea FruityFlix
## 10 1971 Sì Diploma di scuola superiore Coal Kings
## # i 990 more rows

data |>
  select(-c(gender, edu))

## # A tibble: 1,000 x 3
##   birth.y nato.it company
##   <dbl> <chr> <chr>
## 1 1994 Sì My Vegetarian Dinner
## 2 1968 No Urban Gallery
## 3 1969 Sì Office Tile
## 4 1964 Sì Raven
## 5 1971 Sì Fix Guru
## 6 1961 Sì Office Brush
## 7 1993 Sì House Brush
## 8 1981 Sì Satan's Sister
## 9 1979 Sì FruityFlix
## 10 1971 Sì Coal Kings
## # i 990 more rows

# Reorder variables.
data |>
  select(company, edu, gender)

## # A tibble: 1,000 x 3
##   company edu gender
##   <chr> <chr> <chr>
## 1 My Vegetarian Dinner Master post-laurea Uomo
## 2 Urban Gallery Laurea Donna
## 3 Office Tile Diploma di scuola superiore Uomo
## 4 Raven Diploma di scuola superiore Uomo
## 5 Fix Guru Diploma di scuola superiore Uomo

```

```
## 6 Office Brush      Laurea      Preferisco non r~
## 7 House Brush      Scuola di specializzazione post-laurea Donna
## 8 Satan's Sister   Laurea      Uomo
## 9 FruityFlix       Laurea      Donna
## 10 Coal Kings      Diploma di scuola superiore Uomo
## # i 990 more rows
```

```
# Subset/reorder while renaming
data |>
  select(company2 = company, education = edu)
```

```
## # A tibble: 1,000 x 2
##   company2      education
##   <chr>        <chr>
## 1 My Vegetarian Dinner Master post-laurea
## 2 Urban Gallery    Laurea
## 3 Office Tile      Diploma di scuola superiore
## 4 Raven            Diploma di scuola superiore
## 5 Fix Guru         Diploma di scuola superiore
## 6 Office Brush     Laurea
## 7 House Brush      Scuola di specializzazione post-laurea
## 8 Satan's Sister   Laurea
## 9 FruityFlix       Laurea
## 10 Coal Kings      Diploma di scuola superiore
## # i 990 more rows
```

```
# Select offers very flexible syntax to indicate variable names, for example:
# Select variables whose name ends with "y".
data |>
  select(ends_with("y"))
```

```
## # A tibble: 1,000 x 2
##   birth.y company
##   <dbl> <chr>
## 1 1994 My Vegetarian Dinner
## 2 1968 Urban Gallery
## 3 1969 Office Tile
## 4 1964 Raven
## 5 1971 Fix Guru
## 6 1961 Office Brush
## 7 1993 House Brush
## 8 1981 Satan's Sister
## 9 1979 FruityFlix
## 10 1971 Coal Kings
## # i 990 more rows
```

```
# Select variables which are numeric.
data |>
```

```
select(where(is.numeric))

## # A tibble: 1,000 x 1
##   birth.y
##   <dbl>
## 1    1994
## 2    1968
## 3    1969
## 4    1964
## 5    1971
## 6    1961
## 7    1993
## 8    1981
## 9    1979
## 10   1971
## # i 990 more rows

# See ?select for all options.

# Note the difference between these two types of subsetting.

# Option 1: keeping the data frame structure
data |>
  select(birth.y)

## # A tibble: 1,000 x 1
##   birth.y
##   <dbl>
## 1    1994
## 2    1968
## 3    1969
## 4    1964
## 5    1971
## 6    1961
## 7    1993
## 8    1981
## 9    1979
## 10   1971
## # i 990 more rows

# This is the same as
data["birth.y"]

## # A tibble: 1,000 x 1
##   birth.y
##   <dbl>
## 1    1994
```

```
## 2    1968
## 3    1969
## 4    1964
## 5    1971
## 6    1961
## 7    1993
## 8    1981
## 9    1979
## 10   1971
## # i 990 more rows
```

Option 2: getting the inner vector out of the data frame structure (just print the first few values).

```
data |>
  pull(birth.y) |>
  head()
```

```
## [1] 1994 1968 1969 1964 1971 1961
```

This is the same as

```
data$birth.y |>
  head()
```

```
## [1] 1994 1968 1969 1964 1971 1961
```

3.2 Recoding, cleaning, creating variables

- The `mutate()` verb.
- Creating new continuous (i.e., interval or ratio) variables.
- Recoding continuous variables into categorical (i.e., ordinal or nominal), aka factors.
- Recoding factors: relabeling factor levels (i.e., categories).
- Reordering factor levels.
- Recoding factors: combining multiple levels into one.

Create new variable from existing continuous variable.

```
data <- data |>
  mutate(
    age = 2023 - birth.y
  )
```

See result.

```
data |>
  select(birth.y, age)
```

```
## # A tibble: 1,000 x 2
##   birth.y age
##   <dbl> <dbl>
```



```
## 1 1994 29
## 2 1968 55
## 3 1969 54
## 4 1964 59
## 5 1971 52
## 6 1961 62
## 7 1993 30
## 8 1981 42
## 9 1979 44
## 10 1971 52
## # i 990 more rows

# Recode continuous variable into categorical (i.e., factor).

# Age -> Generation
# Gen Z: 1996-2010 -> Fascia età 13-27
# Millennials: 1980-1995 -> Fascia età 28-43
# Gen X: 1965-1979 -> Fascia età 44-58
# Baby boomers: 1946-1964 -> Fascia età 59-77 (ma includiamo respondents fino a
# 80).
data <- data |>
  mutate(
    gen1 = cut(age, c(13, 27, 43, 58, 80))
  )

# View. Note the new variable is listed as <fct>.
data |>
  select(birth.y, age, gen1)

## # A tibble: 1,000 x 3
##   birth.y   age gen1
##   <dbl> <dbl> <fct>
## 1 1994    29 (27,43]
## 2 1968    55 (43,58]
## 3 1969    54 (43,58]
## 4 1964    59 (58,80]
## 5 1971    52 (43,58]
## 6 1961    62 (58,80]
## 7 1993    30 (27,43]
## 8 1981    42 (27,43]
## 9 1979    44 (43,58]
## 10 1971    52 (43,58]
## # i 990 more rows

# Recode factor: new labels for generations
data <- data |>
  mutate(
```

```

    gen = fct_recode(gen1, `gen z` = "(13,27]", `millennials` = "(27,43]",
                     `gen x` = "(43,58]", `baby boomers` = "(58,80]")
  )

# View.
data |>
  select(age, gen1, gen)

## # A tibble: 1,000 x 3
##   age gen1      gen
##   <dbl> <fct>   <fct>
## 1    29 (27,43] millennials
## 2    55 (43,58] gen x
## 3    54 (43,58] gen x
## 4    59 (58,80] baby boomers
## 5    52 (43,58] gen x
## 6    62 (58,80] baby boomers
## 7    30 (27,43] millennials
## 8    42 (27,43] millennials
## 9    44 (43,58] gen x
## 10   52 (43,58] gen x
## # i 990 more rows

# Factors have levels (i.e., categories). These are stored in a specific order.
data |>
  pull(gen) |>
  levels()

## [1] "gen z"      "millennials" "gen x"      "baby boomers"

# This order affects the output of certain functions, including those for
# producing summary tables and visualizations. For example:
data |>
  tabyl(gen)

##           gen    n percent
##      gen z    17   0.017
## millennials 297   0.297
##      gen x   576   0.576
## baby boomers 110   0.110

# We can change the order of levels.
data <- data |>
  mutate(
    gen = fct_relevel(gen, "baby boomers", "gen x", "millennials", "gen z")
  )

```

```

# See the result.
data |>
  pull(gen) |>
  levels()

## [1] "baby boomers" "gen x"          "millennials" "gen z"

# Now the summary table has rows in a different order.
data |>
  tabyl(gen)

##           gen    n percent
## baby boomers 110   0.110
##          gen x 576   0.576
## millennials 297   0.297
##          gen z  17   0.017

# Recode age in a different way: 10-year age brackets.
data <- data |>
  mutate(
    age.br = cut(age, c(20, 30, 40, 50, 60, 70, 80))
  )

# View.
data |>
  select(age, age.br)

## # A tibble: 1,000 x 2
##       age age.br
##   <dbl> <fct>
## 1    29 (20,30]
## 2    55 (50,60]
## 3    54 (50,60]
## 4    59 (50,60]
## 5    52 (50,60]
## 6    62 (60,70]
## 7    30 (20,30]
## 8    42 (40,50]
## 9    44 (40,50]
## 10   52 (50,60]
## # i 990 more rows

# Levels of the new variable
data |>
  pull(age.br) |>
  levels()

## [1] "(20,30]" "(30,40]" "(40,50]" "(50,60]" "(60,70]" "(70,80]"

```

```
# Recode, i.e., re-label age brackets.
```

```
data <- data |>
  mutate(
    age.br.2 = fct_recode(age.br,
      `21-30` = "(20,30]",
      `31-40` = "(30,40]",
      `41-50` = "(40,50]",
      `51-60` = "(50,60]",
      `61-70` = "(60,70]",
      `71-80` = "(70,80]"
    )
  )
```

```
# View results.
```

```
data |>
  select(age, age.br, age.br.2)
```

```
## # A tibble: 1,000 x 3
##       age age.br age.br.2
##   <dbl> <fct>   <fct>
## 1    29 (20,30] 21-30
## 2    55 (50,60] 51-60
## 3    54 (50,60] 51-60
## 4    59 (50,60] 51-60
## 5    52 (50,60] 51-60
## 6    62 (60,70] 61-70
## 7    30 (20,30] 21-30
## 8    42 (40,50] 41-50
## 9    44 (40,50] 41-50
## 10   52 (50,60] 51-60
## # i 990 more rows
```

```
# Collapse 10-year brackets into 20-year.
```

```
data <- data |>
  mutate(
    age.br.3 = fct_collapse(age.br.2,
      `21-40` = c("21-30", "31-40"),
      `41-60` = c("41-50", "51-60"),
      `61-80` = c("61-70", "71-80")
    )
  )
```

```
# View results.
```

```
data |>
  select(age, age.br.2, age.br.3)
```

```
## # A tibble: 1,000 x 3
##   age age.br.2 age.br.3
##   <dbl> <fct>   <fct>
## 1    29 21-30    21-40
## 2    55 51-60    41-60
## 3    54 51-60    41-60
## 4    59 51-60    41-60
## 5    52 51-60    41-60
## 6    62 61-70    61-80
## 7    30 21-30    21-40
## 8    42 41-50    41-60
## 9    44 41-50    41-60
## 10   52 51-60    41-60
## # i 990 more rows
```

```
# Check consistency with cross-tabulations.
```

```
data |>
  tabyl(age.br.2)
```

```
##   age.br.2    n percent
##   21-30    58   0.058
##   31-40   175   0.175
##   41-50   325   0.325
##   51-60   391   0.391
##   61-70    50   0.050
##   71-80     1   0.001
```

```
data |>
  tabyl(age.br.3)
```

```
##   age.br.3    n percent
##   21-40   233   0.233
##   41-60   716   0.716
##   61-80    51   0.051
```

```
# Is this what we should expect?
```

```
data |>
  tabyl(age.br.2, age.br.3)
```

```
##   age.br.2 21-40 41-60 61-80
##   21-30    58     0     0
##   31-40   175     0     0
##   41-50     0   325     0
##   51-60     0   391     0
##   61-70     0     0    50
##   71-80     0     0     1
```

```
# Another example of recoding/collapsing. Check out the original education
# categories (are they in the correct order?).
```

```
data |>
  tabyl(edu)
```

```
##              edu    n percent
##      Diploma di scuola superiore 416    0.416
##      Diploma di scuole medie o inferiore 23    0.023
##              Dottorato 17    0.017
##              Laurea 359    0.359
##      Master post-laurea 127    0.127
##      Scuola di specializzazione post-diploma 31    0.031
##      Scuola di specializzazione post-laurea 27    0.027
```

```
# Recode education. Note other_level = ... argument.
```

```
data <- data |>
  mutate(
    edu = fct_collapse(edu,
      `Medie o meno` = "Diploma di scuole medie o inferiore",
      Superiori = "Diploma di scuola superiore",
      Laurea = "Laurea",
      `Post-laurea` = c("Master post-laurea", "Dottorato"),
      other_level = "Altro"
    )
  )
```

```
# Results:
```

```
data |>
  tabyl(edu)
```

```
##              edu    n percent
##      Superiori 416    0.416
##      Medie o meno 23    0.023
##      Post-laurea 144    0.144
##      Laurea 359    0.359
##      Altro 58    0.058
```

```
# We still need to arrange levels (categories) in the right order.
```

```
data <- data |>
  mutate(
    edu = fct_relevel(edu, "Medie o meno", "Superiori", "Laurea", "Post-laurea", "Altro")
  )
```

```
# See the new order of levels.
```

```
data |>
  tabyl(edu)
```

```
##          edu    n percent
##  Medie o meno  23    0.023
##    Superiori 416    0.416
##      Laurea 359    0.359
## Post-laurea 144    0.144
##        Altro  58    0.058
```

```
# Recode gender.
```

```
# See unique (distinct) values of gender
```

```
data |>
  tabyl(gender)
```

```
##                                gender    n percent
## Altra identità di genere (transgender, non binario, ecc.)    3    0.003
##                                Donna 522    0.522
##                                Preferisco non rispondere    8    0.008
##                                Uomo 467    0.467
```

```
# Recode and reorder levels in the same call
```

```
data <- data |>
```

```
  mutate(
    gender = fct_recode(gender,
                        NR = "Preferisco non rispondere",
                        `Altra identità` = "Altra identità di genere (transgender, non binario, ecc.)",
                        fct_relevel("Donna", "Uomo", "Altra identità")
    )
```

```
# View
```

```
data |>
  tabyl(gender)
```

```
##          gender    n percent
##      Donna 522    0.522
##      Uomo 467    0.467
## Altra identità    3    0.003
##          NR    8    0.008
```

```
# Only keep variables of interest for the following.
```

```
# What is this code doing with the age.br variable?
```

```
data <- data |>
```

```
  select(birth.y, age, gen, age.br = age.br.2, gender, nato.it, edu, company)
```

3.3 Filtering and rearranging cases

- The `filter()` verb.
- Filtering cases (i.e., data rows) based on logical conditions with relational

operators.

- Combining multiple logical conditions via intersection (&) or union (|).
- Rearranging data rows.

Respondents whose age is exactly 40.

```
data |>
  filter(age == 40)
```

```
## # A tibble: 26 x 8
##   birth.y   age gen          age.br gender nato.it edu      company
##   <dbl> <dbl> <fct>      <fct> <fct> <chr> <fct> <chr>
## 1  1983    40 millennials 31-40 Uomo   Sì     Altro  Fix Guru
## 2  1983    40 millennials 31-40 Uomo   Sì     Superiori Bloom Marketing
## 3  1983    40 millennials 31-40 Donna  Sì     Post-laurea Urban Philosophy
## 4  1983    40 millennials 31-40 Donna  Sì     Laurea  Wood Works
## 5  1983    40 millennials 31-40 Uomo   Sì     Laurea  Office Tile
## 6  1983    40 millennials 31-40 Uomo   Sì     Laurea  The Spice Route
## 7  1983    40 millennials 31-40 Uomo   Sì     Superiori The Zone
## 8  1983    40 millennials 31-40 Uomo   Sì     Superiori Coal Kings
## 9  1983    40 millennials 31-40 Uomo   <NA>    Altro  FruityFlix
## 10 1983    40 millennials 31-40 Donna  Sì     Superiori Raven
## # i 16 more rows
```

With education = Laurea

```
data |>
  filter(edu == "Laurea")
```

```
## # A tibble: 359 x 8
##   birth.y   age gen          age.br gender nato.it edu      company
##   <dbl> <dbl> <fct>      <fct> <fct> <chr> <fct> <chr>
## 1  1968    55 gen x      51-60 Donna  No     Laurea Urban Gallery
## 2  1961    62 baby boomers 61-70 NR     Sì     Laurea Office Brush
## 3  1981    42 millennials 41-50 Uomo   Sì     Laurea Satan's Sister
## 4  1979    44 gen x      41-50 Donna  Sì     Laurea FruityFlix
## 5  1975    48 gen x      41-50 Donna  Sì     Laurea Coal Kings
## 6  1978    45 gen x      41-50 Uomo   Sì     Laurea The Auto DNA
## 7  1972    51 gen x      51-60 Uomo   Sì     Laurea Wood Works
## 8  1987    36 millennials 31-40 Donna  Sì     Laurea Garden Store
## 9  1984    39 millennials 31-40 Uomo   Sì     Laurea FruityFlix
## 10 1981    42 millennials 41-50 Uomo   Sì     Laurea Wood Works
## # i 349 more rows
```

The %in% operator is useful when you need to select multiple values.

```
data |>
  filter(edu %in% c("Superiori", "Laurea"))
```

```
## # A tibble: 775 x 8
##   birth.y   age gen          age.br gender nato.it edu      company
```



```
##      <dbl> <dbl> <fct>      <fct> <fct> <chr>      <fct>      <chr>
## 1    1968    55 gen x      51-60 Donna No      Laurea      Urban Gallery
## 2    1969    54 gen x      51-60 Uomo  Si      Superiori Office Tile
## 3    1964    59 baby boomers 51-60 Uomo  Si      Superiori Raven
## 4    1971    52 gen x      51-60 Uomo  Si      Superiori Fix Guru
## 5    1961    62 baby boomers 61-70 NR    Si      Laurea      Office Brush
## 6    1981    42 millennials 41-50 Uomo  Si      Laurea      Satan's Sister
## 7    1979    44 gen x      41-50 Donna Si      Laurea      FruityFlix
## 8    1971    52 gen x      51-60 Uomo  Si      Superiori Coal Kings
## 9    1966    57 gen x      51-60 Uomo  Si      Superiori Coal Kings
## 10   1975    48 gen x      41-50 Donna Si      Laurea      Coal Kings
## # i 765 more rows
```

```
# All education categories except "Medie o meno"
data |>
  filter(edu != "Medie o meno")
```

```
## # A tibble: 977 x 8
##   birth.y age gen      age.br gender nato.it edu      company
##   <dbl> <dbl> <fct>      <fct> <fct> <chr>      <fct>      <chr>
## 1    1994    29 millennials 21-30 Uomo  Si      Post-laurea My Vegetarian D~
## 2    1968    55 gen x      51-60 Donna No      Laurea      Urban Gallery
## 3    1969    54 gen x      51-60 Uomo  Si      Superiori Office Tile
## 4    1964    59 baby boomers 51-60 Uomo  Si      Superiori Raven
## 5    1971    52 gen x      51-60 Uomo  Si      Superiori Fix Guru
## 6    1961    62 baby boomers 61-70 NR    Si      Laurea      Office Brush
## 7    1993    30 millennials 21-30 Donna Si      Altro      House Brush
## 8    1981    42 millennials 41-50 Uomo  Si      Laurea      Satan's Sister
## 9    1979    44 gen x      41-50 Donna Si      Laurea      FruityFlix
## 10   1971    52 gen x      51-60 Uomo  Si      Superiori Coal Kings
## # i 967 more rows
```

```
# Missing value on nato.it
data |>
  filter(is.na(nato.it))
```

```
## # A tibble: 4 x 8
##   birth.y age gen      age.br gender nato.it edu      company
##   <dbl> <dbl> <fct>      <fct> <fct> <chr>      <fct>      <chr>
## 1    1967    56 gen x      51-60 Donna <NA>      Superiori Fix Guru
## 2    1967    56 gen x      51-60 Donna <NA>      Superiori Bloom Marketing
## 3    1983    40 millennials 31-40 Uomo  <NA>      Altro      FruityFlix
## 4    1974    49 gen x      41-50 Uomo  <NA>      Superiori Art Fade
```

```
# Multiple conditions can be combined: intersection (&, i.e. AND) or
# union (|, i.e. OR).
```

```
# Respondents between 30 and 40.
```

```
data |>
  filter(age >= 30 & age <=40)
```

```
## # A tibble: 190 x 8
##   birth.y   age gen          age.br gender nato.it edu      company
##   <dbl> <dbl> <fct>      <fct> <fct> <chr>   <fct>   <chr>
## 1   1993    30 millennials 21-30 Donna Si      Altro   House Brush
## 2   1984    39 millennials 31-40 Uomo  Si      Superiori Wood Works
## 3   1990    33 millennials 31-40 Uomo  Si      Post-laurea Art Fade
## 4   1989    34 millennials 31-40 Donna Si      Post-laurea The Loop
## 5   1993    30 millennials 21-30 Donna Si      Post-laurea Service Scientist
## 6   1988    35 millennials 31-40 Donna Si      Superiori FruityFlix
## 7   1987    36 millennials 31-40 Donna Si      Laurea   Garden Store
## 8   1983    40 millennials 31-40 Uomo  Si      Altro    Fix Guru
## 9   1984    39 millennials 31-40 Uomo  Si      Laurea   FruityFlix
## 10  1988    35 millennials 31-40 NR     Si      Post-laurea Wood Works
## # i 180 more rows
```

Respondents who are either younger than 20 or older than 40.

```
data |>
  filter(age < 20 | age > 40)
```

```
## # A tibble: 767 x 8
##   birth.y   age gen          age.br gender nato.it edu      company
##   <dbl> <dbl> <fct>      <fct> <fct> <chr>   <fct>   <chr>
## 1   1968    55 gen x      51-60 Donna No      Laurea   Urban Gallery
## 2   1969    54 gen x      51-60 Uomo  Si      Superiori Office Tile
## 3   1964    59 baby boomers 51-60 Uomo  Si      Superiori Raven
## 4   1971    52 gen x      51-60 Uomo  Si      Superiori Fix Guru
## 5   1961    62 baby boomers 61-70 NR     Si      Laurea   Office Brush
## 6   1981    42 millennials 41-50 Uomo  Si      Laurea   Satan's Sister
## 7   1979    44 gen x      41-50 Donna Si      Laurea   FruityFlix
## 8   1971    52 gen x      51-60 Uomo  Si      Superiori Coal Kings
## 9   1966    57 gen x      51-60 Uomo  Si      Superiori Coal Kings
## 10  1975    48 gen x      41-50 Donna Si      Laurea   Coal Kings
## # i 757 more rows
```

Respondents who are younger than 40 AND whose education is post-laurea

```
data |>
  filter(age < 40 & edu == "Post-laurea")
```

```
## # A tibble: 51 x 8
##   birth.y   age gen          age.br gender nato.it edu      company
##   <dbl> <dbl> <fct>      <fct> <fct> <chr>   <fct>   <chr>
## 1   1994    29 millennials 21-30 Uomo  Si      Post-laurea My Vegetarian Di~
## 2   1990    33 millennials 31-40 Uomo  Si      Post-laurea Art Fade
## 3   1989    34 millennials 31-40 Donna Si      Post-laurea The Loop
```

```
## 4 1993 30 millennials 21-30 Donna Si Post-laurea Service Scientist
## 5 1988 35 millennials 31-40 NR Si Post-laurea Wood Works
## 6 1984 39 millennials 31-40 Donna Si Post-laurea Raven
## 7 1987 36 millennials 31-40 Donna Si Post-laurea The Zone
## 8 1985 38 millennials 31-40 Uomo Si Post-laurea Coal Kings
## 9 1985 38 millennials 31-40 Uomo Si Post-laurea Beep Sports
## 10 1997 26 gen z 21-30 Donna No Post-laurea Satan's Sister
## # i 41 more rows
```

The filtered data can be saved to new objects, for example:

```
data.40 <- data |>
  filter(age <= 40)
```

Different dplyr verbs can be combined in a pipe chain:

```
data |>
  filter(age < 40) |>
  select(age, gen, edu)
```

```
## # A tibble: 207 x 3
##   age gen      edu
##   <dbl> <fct>   <fct>
## 1 29 millennials Post-laurea
## 2 30 millennials Altro
## 3 39 millennials Superiori
## 4 33 millennials Post-laurea
## 5 34 millennials Post-laurea
## 6 30 millennials Post-laurea
## 7 35 millennials Superiori
## 8 36 millennials Laurea
## 9 39 millennials Laurea
## 10 35 millennials Post-laurea
## # i 197 more rows
```

Rearrange rows by increasing values on a variable.

```
data |>
  arrange(birth.y)
```

```
## # A tibble: 1,000 x 8
##   birth.y age gen      age.br gender nato.it edu      company
##   <dbl> <dbl> <fct>   <fct> <fct> <chr>   <fct>   <chr>
## 1 1944 79 baby boomers 71-80 Uomo Si Laurea The Glow Up
## 2 1954 69 baby boomers 61-70 NR Si Medie o meno Brunchies
## 3 1958 65 baby boomers 61-70 Uomo Si Laurea Raven
## 4 1958 65 baby boomers 61-70 Donna Si Laurea The Spice Route
## 5 1958 65 baby boomers 61-70 Uomo Si Laurea The Zone
## 6 1958 65 baby boomers 61-70 Uomo Si Laurea Raven
## 7 1958 65 baby boomers 61-70 Uomo Si Laurea Evergrow
```

```
## 8      1958      65 baby boomers 61-70 Donna Si      Superiori Coal Kings
## 9      1958      65 baby boomers 61-70 Uomo  Si      Altro    Raven
## 10     1959      64 baby boomers 61-70 Donna Si      Laurea   The Zone
## # i 990 more rows
```

```
# By decreasing value:
data |>
  arrange(desc(birth.y))
```

```
## # A tibble: 1,000 x 8
##   birth.y age gen age.br gender nato.it edu      company
##   <dbl> <dbl> <fct> <fct> <fct> <chr> <fct> <chr>
## 1     2002    21 gen z 21-30 Uomo  Si      Superiori Fix Guru
## 2     2000    23 gen z 21-30 Donna Si      Laurea   My Plumber
## 3     2000    23 gen z 21-30 Uomo  Si      Superiori Coal Kings
## 4     2000    23 gen z 21-30 Uomo  Si      Superiori Coal Kings
## 5     2000    23 gen z 21-30 Donna Si      Laurea   Beep Sports
## 6     2000    23 gen z 21-30 Uomo  Si      Laurea   Raven
## 7     1998    25 gen z 21-30 Donna Si      Laurea   The Fresh Breakfast
## 8     1998    25 gen z 21-30 Donna Si      Laurea   My Vegetarian Dinner
## 9     1997    26 gen z 21-30 Donna No      Post-laurea Satan's Sister
## 10    1997    26 gen z 21-30 Uomo  Si      Laurea   Art Fade
## # i 990 more rows
```

3.4 Describing or summarizing the data

3.4.1 With `dplyr::summarize()`

- The `summarize()` function takes and applies it to one or multiple variables. We can use existing R functions or any new function we have defined.
- Remember what `sum()` and `mean()` do to a logical vector. We can use this to calculate absolute or relative frequencies of specific levels (categories) of factors.

```
# Mean of one continuous variable
data |>
  summarize(avg.age = mean(age))
```

```
## # A tibble: 1 x 1
##   avg.age
##   <dbl>
## 1     47.6
```

```
# Number of unique or distinct values in edu.
data |>
  summarize(n.edu = n_distinct(edu))
```

```
## # A tibble: 1 x 1
```

```
##   n.edu
##   <int>
## 1     5

# Absolute frequency of women.
data |>
  summarize(count.women = sum(gender == "Donna"))

## # A tibble: 1 x 1
##   count.women
##         <int>
## 1         522

# We can obtain a battery of multiple summary statistics.
data |>
  summarize(
    min.year = min(birth.y),
    avg.age = mean(age),
    sd.age = sd(age),
    count.women = sum(gender == "Donna"),
    # What am I doing in this last line?
    prop.foreign = mean(nato.it == "No", na.rm = TRUE)
  )

## # A tibble: 1 x 5
##   min.year avg.age sd.age count.women prop.foreign
##   <dbl>   <dbl> <dbl>         <int>         <dbl>
## 1   1944   47.6   9.39           522         0.0221
```

3.4.2 With specific functions for descriptive analysis

- `skimr` for descriptive statistics on continuous variable
- `janitor` for tabulations of categorical variables.

```
# Battery of descriptive statistics on age.
data |>
  skimr::skim_tee(age)
```

```
## -- Data Summary -----
##                               Values
## Name                         data
## Number of rows                1000
## Number of columns             8
## -----
## Column type frequency:
##   numeric                     1
## -----
## Group variables              None
```

```
##
## -- Variable type: numeric -----
##   skim_variable n_missing complete_rate mean   sd p0  p25 p50 p75 p100 hist
## 1 age           0              1 47.6 9.39 21 41.8 49 55 79

# Tabulation for gender
data |>
  janitor::tabyl(gender)

##           gender    n percent
##           Donna 522   0.522
##           Uomo  467   0.467
## Altra identità   3   0.003
##           NR    8   0.008

# Note that count() gives us similar information.
data |>
  count(gender)

## # A tibble: 4 x 2
##   gender      n
##   <fct>    <int>
## 1 Donna    522
## 2 Uomo     467
## 3 Altra identità  3
## 4 NR       8

# Cross-tabulation for education and gender
data |>
  tabyl(edu, gender)

##           edu Donna Uomo Altra identità NR
## Medie o meno    5   15          0  3
## Superiori     196  218          1  1
## Laurea        207  150          0  2
## Post-laurea    85   57          1  1
## Altro          29   27          1  1
```

- `janitor` offers different ways of customizing tabulations.

```
# Add column totals.
data |>
  tabyl(gender) |>
  adorn_totals(where = "row")

##           gender    n percent
##           Donna 522   0.522
##           Uomo  467   0.467
## Altra identità   3   0.003
```

```
##          NR      8    0.008
##        Total 1000    1.000
```

```
# Format percentage
data |>
  tabyl(gender) |>
  adorn_totals(where = "row") |>
  adorn_pct_formatting()
```

```
##          gender    n percent
##        Donna   522   52.2%
##        Uomo   467   46.7%
## Altra identità    3    0.3%
##          NR      8    0.8%
##        Total 1000  100.0%
```

- Note that both `skimr` and `janitor` output objects can be converted to tibbles for further manipulation.

```
data |>
  skimr::skim(age) |>
  as_tibble() |>
  select(variable = skim_variable, mean = numeric.mean, sd = numeric.sd)
```

```
## # A tibble: 1 x 3
##   variable mean    sd
##   <chr>    <dbl> <dbl>
## 1 age      47.6  9.39
```

3.4.3 Descriptive statistics by group

- The `group_by()` function allows us to run any descriptive analysis by data subsets.
- These subsets may come from the categories of a single factor or a combination of categories from multiple factors.
- Both `summarize()` and `skimr` are compatible with `group_by()`

Let's run `summarize()` as above, but now by gender. What are the results telling us?

```
data |>
  group_by(gender) |>
  summarize(
    min.year = min(birth.y),
    avg.age = mean(age),
    sd.age = sd(age),
    prop.foreign = mean(nato.it == "No", na.rm = TRUE)
  )
```

```
## # A tibble: 4 x 5
##   gender    min.year avg.age sd.age prop.foreign
```

```
##   <fct>           <dbl>  <dbl>  <dbl>      <dbl>
## 1 Donna           1958    46.7   9.07      0.025
## 2 Uomo            1944    48.6   9.64      0.0194
## 3 Altra identità  1964    48     9.64      0
## 4 NR              1954    50.6  11.1      0
```

```
# Average age in each company
data |>
  group_by(company) |>
  summarize(avg.age = mean(age))
```

```
## # A tibble: 53 x 2
##   company      avg.age
##   <chr>        <dbl>
## 1 Active Body    48.1
## 2 Art Fade      44.2
## 3 Bean Morning  50.1
## 4 Beep Sports   43.1
## 5 Bloom Marketing 47.2
## 6 Brew Bean     35
## 7 Brunchies     69
## 8 Clean Scissor  46
## 9 Coal Kings    52.3
## 10 Death By Milkshake 43.8
## # i 43 more rows
```

```
# Let's now only keep Donna and Uomo as gender, and re-run by groups given by
# combinations of gender and education categories.
```

```
data |>
  filter(gender %in% c("Donna", "Uomo")) |>
  group_by(gender, edu) |>
  summarize(
    min.year = min(birth.y),
    avg.age = mean(age),
    sd.age = sd(age),
    prop.foreign = mean(nato.it == "No", na.rm = TRUE)
  )
```

```
## `summarise()` has grouped output by 'gender'. You can override using the
## `.groups` argument.
```

```
## # A tibble: 10 x 6
## # Groups:   gender [2]
##   gender edu      min.year avg.age sd.age prop.foreign
##   <fct> <fct>        <dbl>  <dbl>  <dbl>      <dbl>
## 1 Donna Medie o meno    1960    53.6   7.80      0
## 2 Donna Superiori      1958    50.1   7.13     0.0206
## 3 Donna Laurea         1958    44.9   9.81     0.0290
```



```
## 4 Donna Post-laurea 1963 43.3 8.63 0.0353
## 5 Donna Altro 1964 45.7 9.06 0
## 6 Uomo Medie o meno 1964 52.9 4.89 0.0667
## 7 Uomo Superiori 1959 51.7 8.19 0.0276
## 8 Uomo Laurea 1944 45.1 10.2 0.00667
## 9 Uomo Post-laurea 1963 43 9.24 0.0175
## 10 Uomo Altro 1958 52.6 7.83 0
```

skimr functions are also compatible with group_by().

```
data |>
  group_by(gender) |>
  skim_tee(age)
```

```
## -- Data Summary -----
##                               Values
## Name                         data
## Number of rows               1000
## Number of columns            8
## -----
## Column type frequency:
##   numeric                     1
## -----
## Group variables              gender
##
## -- Variable type: numeric -----
##   skim_variable gender      n_missing complete_rate mean    sd p0  p25 p50
## 1 age          Donna         0             1 46.7  9.07 23 41  48
## 2 age          Uomo         0             1 48.6  9.64 21 42  51
## 3 age          Altra identità 0             1 48    9.64 41 42.5 44
## 4 age          NR           0             1 50.6 11.1 35 44  49
##   p75 p100 hist
## 1 54    65
## 2 56    79
## 3 51.5  59
## 4 56.8  69
```

3.4.4 Export results

- Descriptive results from `summarize()`, `skimr` or `janitor` can be exported to external file (e.g., csv). In a later session we'll see how to directly output them into a report (e.g. in html or pdf).

```
data |>
  filter(gender %in% c("Donna", "Uomo")) |>
  group_by(gender, edu) |>
  summarize(
    min.year = min(birth.y),
```

```
avg.age = mean(age),  
sd.age = sd(age),  
prop.foreign = mean(nato.it == "No", na.rm = TRUE)  
) |>  
write_csv("summarize_by_gender-edu.csv")
```

`summarise()` has grouped output by 'gender'. You can override using the
`.groups` argument.

```
data |>  
  group_by(gender) |>  
  skim(age) |>  
  as_tibble() |>  
  write_csv("skim_by_gender.csv")
```

Chapter 4

Data visualization

Chapter 5

Creating reproducible reports