# Introduction to the R programming language

**Data visualization 2023/2024**

Matteo Ceccarello

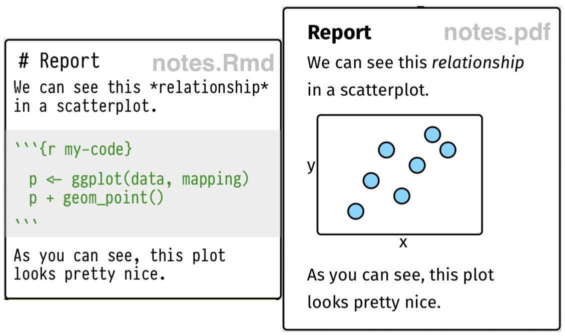
2025-01-12

In this course we use the R programming language, which provides powerful plotting facilities. You can code with the convenience of an Integrated Development Environment without installing anything on your computer by using the online IDE provided at https://posit.cloud.

In most of this course we will use either plain R scripts or notebooks written with Quarto. Quarto is *an open-source scientific and technical publishing system* that allows to render plain-text notebooks in a variety of formats.

These lecture notes are written using Quarto, which allows to interweave text with code and its output, even graphical output.

The general pipeline is reported in the margin figure. We will write text and code in a *plain-text* file (with extension `.qmd` for `Quarto Markdown`). A Quarto Markdown document can be *rendered* to a variety of formats such as HTML or PDF. During the process of rendering, specially marked *code blocks* are executed, and their output is injected in the rendered document, possibly replacing the code that was used to compute the output.



~~~
# Report
We can see this *relationship*
in a scatterplot.

```{r my-code}
  p <- ggplot(data, mapping)
  p + geom_point()
```

As you can see, this plot
looks pretty nice.
~~~

For reference on how to write Quarto Markdown documents, you can refer to the official documentation at https://quarto.org.

## A brief overview of the R programming language

We will now cover a few key aspects of the R programming language. For the purposes of this course we will not need to become experts in the language. For a more comprehensive introduction to the R programming language Kieran Healy's book[1] is a good starting point.

### Data types

R has five basic data types:

- `character`

  ```
  "hello"
  ```

- `numeric` (a.k.a. real numbers, `double`)

  ```
  0.82
  ```

[1] *Data Visualization. A practical introcution.* Kieran Healy. https://socviz.co/gettingstarted.html

- `integer`

  ```
  42L
  ```

- `complex`

  ```
  3.2 + 5.2i
  ```

- `logical` (a.k.a. booleans)

  ```
  TRUE   # same as T
  ```

  ```
  F      # Same as FALSE
  ```

### Names

In R, everything can be given a name

This is a valid name:

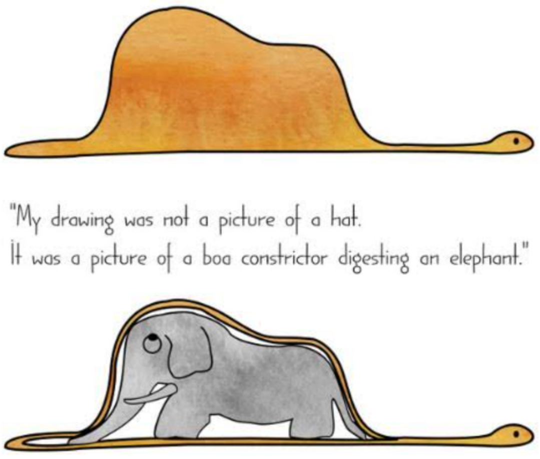
```
x
```

Descriptive names are preferable

- note the underscore separating the words
- spaces are not allowed

```
descriptive_name
```

This is often referred to as `snake_case` naming.


"My drawing was not a picture of a hat.
It was a picture of a boa constrictor digesting an elephant."

#

The following is also a valid name, using an older and maybe confusing naming scheme. If you come from Java/C++/Python/Javascript…. the . in the middle of the name is *not* the member access operator

```
also.valid
```

The above naming scheme is used in R's standard library, whereas the snake case convention is used in the `tidyverse` family of library that we will use in this course.

The following names are not allowed, as they are reserved keywords.

```
FALSE, TRUE, Inf, NA, NaN, NULL,
for, if, else, break, function
```

Some names are best avoided, because they are library functions that you would overwrite

```
mean, range
```

You can use them and the R interpreter will not complain[2] but most likely one does not want to overwrite the built in functions.

[2] Did we mention that R is an interpreted language?

### Binding things to names

Using the "arrow" syntax you can assign names to things

```r
x <- 5  # The `arrow` is the assignment operator
some_string <- "Hello I am a sequence of characters"
```

Later on, you can retrieve the values by simply referencing the name

Note that, in these lecture notes, oftentimes the result of the evaluation of a code block is displayed right after the code block. This holds true for graphical output as well.

```r
x
```

```
[1] 5
```

```r
some_string
```

```
[1] "Hello I am a sequence of characters"
```

### Using R as a calculator

Arithmetic

```r
# addition, subtraction, multiplication, division
  +         -             *                /

# quotient, remainder,   power
  %/%       %%            ^
```

Comparisons

```r
<  >   <=  >=  ==  !=
```

Logical (aka boolean operations)

```r
# NOT
  !
# short-circuited AND    short-circuited OR   <- for control flow
  &&                     ||
# AND                    OR                   <- for logical operations
  &                      |
```

! Important

What does the following comparison return (`sqrt` gives the square root)? Is the output of the following operation what you would expect?

```
sqrt(2)^2 == 2
```

```
[1] FALSE
```

`numeric` data is insidious, and comparisons should be handled with care.

```
sqrt(2)^2 - 2
```

```
[1] 4.440892e-16
```

In fact, `numeric` data still uses a finite number of bits, hence it cannot represent the infinite real numbers that lie in any closed interval.
If you want to compare `numeric` values for equality, you can use the following function (more on the `dplyr::` syntax later).

```
dplyr::near(sqrt(2)^2, 2)
```

```
[1] TRUE
```

### Missing values

The `NA` keyword represents a missing value. It "contaminates" any computation it is involved in, making the result `NA`.

```
NA > 3
```

```
[1] NA
```

```
NA + 10
```

```
[1] NA
```

Crucially, this also holds for comparisons with `NA` itself

```
NA == NA
```

```
[1] NA
```

But then how can one check if a value is missing? To check if a value is `NA` you use the `is.na` function

```
a <- NA
is.na(a)
```

```
[1] TRUE
```

```r
b <- "this variable has a value"
is.na(b)
```

```
[1] FALSE
```

### Other special values

What is the result of this operation?

```r
0 / 0
```

```
[1] NaN
```

The `NaN` value (`Not a Number`): the result cannot be represented by a computer.

What about this operation?

```r
sqrt(-1)
```

```
Warning in sqrt(-1): NaNs produced
```

```
[1] NaN
```

We get `NaN` even if this would be the definition of the complex number `i`.

If you want the complex number, then you should declare it explicitly

```r
sqrt( as.complex(-1) )
```

```
[1] 0+1i
```

### `NA` vs `NaN`

**Beware**: in R the values `NA` and `NaN` refer to *distinct* concepts. This is in contrast with Python, where `NaN` is often used also to indicate missing values.

In particular, and confusingly

```r
is.na(NaN)
```

```
[1] TRUE
```

but

```r
is.nan(NA)
```

```
[1] FALSE
```

### Other special values

What about this operation?

```
1 / 0
```

```
[1] Inf
```

The `Inf` value is used to represent infinity, and propagates in calculations

```
Inf + 10
```

```
[1] Inf
```

```
min(Inf, 10)
```

```
[1] 10
```

```
Inf - Inf
```

```
[1] NaN
```

### Vectors

*Atomic* vectors are *homogeneous* indexed collections of values of the *same* basic data type.

```
vec_numbers <- vector("numeric", 4)
vec_numbers
```

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

```
vec_letters <- vector("character", 6)
vec_letters
```

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

You can also define a sequence of numbers with the following syntax

```
1:10
```

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

You can ask for the *type* of a vector using `typeof`

```
typeof(vec_numbers)
```

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

```
typeof(vec_letters)
```

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

```r
typeof(1:10)
```

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

You can ask for the *length* of a vector using `length`

```r
length(vec_numbers)
```

```
[1] 4
```

```r
length(vec_letters)
```

```
[1] 6
```

```r
length(1:10)
```

```
[1] 10
```

What about scalars? What does this return?

```r
typeof(3)
```

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

What about this?

```r
length(3)
```

```
[1] 1
```

There are no scalar values, but vectors of length 1!

The `c` function combines its argments into a vector

```r
c(1, 5, 3, 6, 3)
```

```
[1] 1 5 3 6 3
```

Using `c` multiple times *does not* nest vectors

```r
nums_a <- c(1,3,5,7)
nums_b <- c(2,4,6,8)
c(nums_a, nums_b)
```

```
[1] 1 3 5 7 2 4 6 8
```

Consider now the following snippet

```r
c(1, "hello", 0.45)
```

```
[1] "1"     "hello" "0.45"
```

We said that a vector should have *homogeneously typed* elements. So what is the type of the above vector?

```r
typeof(c(1, "hello", 0.45))
```

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

This is called *implicit coercion* and converts all the elements to the type that can represent all of them

### Coercion

Sometimes automatic implicit yields meaningful results

```r
42L + 3.3
```

```
[1] 45.3
```

Other times it gives errors because the coerced data types do not support the requested operations

```r
3 + "I'm a stringy string"
```

```
Error in 3 + "I'm a stringy string": non-numeric argument to binary operator
```

```r
"ahahaha" & T
```

```
Error in "ahahaha" & T: operations are possible only for numeric, logical or complex types
```

### Recycling

Consider the following operation:"

```r
c(1, 2, 3) + 1
```

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

R coerces the *length* of vectors, if needed[3].

[3] This is similar to *broadcasting* in the Python `numpy` library

Remember that `1` is a vector of length one. By coercion, in the operation above, it is replaced with `c(1, 1, 1)` by *recycling* its value.

The following operation instead results in a warning, because the lengths of the vectors are not a multiple one of the other.

```r
c(1, 2, 3) + c(1, 3)
```

```
Warning in c(1, 2, 3) + c(1, 3): longer object length is not a multiple of
shorter object length
```

```
[1] 2 5 4
```

### Operations on `logical` vectors

There are distinct operators for element-wise operators on logical vectors:

```r
c(T, T, F) & c(T, F, T)
```

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

which is different from

```r
c(T, T, F) && c(T, F, T)
```

```
Error in c(T, T, F) && c(T, F, T): 'length = 3' in coercion to 'logical(1)'
```

If you want to check if all the values are true in a vector, you can use the `all` function:

```r
all(c(T, T, T))
```

```
[1] TRUE
```

or the `any` function to check if at least one value is true

```r
any(c(F, T, F))
```

```
[1] TRUE
```

To check if all the values are false, you can negate the vector

```r
lgls <- c(F, F, F)
all(!lgls)
```

```
[1] TRUE
```

### Naming vectors

Elements of vectors can be named, which will be useful for indexing into the vector

```r
named_vec <- c(
  Alice          = "swimming",
  Bob            = "playing piano",
  Christine      = "cooking",
  Daniel         = "singing",
  "Most people" = "eating"
)
```

Notice that you need to enclose a name in quotes only if it contains spaces. Also, notice that the values in the vector still need to have all the same type.

### Subsetting vectors

You can index into vectors using integers indexes.

**Beware: indexing starts from 1!**

```r
myvec <- c("these", "are", "some", "values")
myvec[3]
```

```
[1] "some"
```

If you ask for out of bounds indices you get different behaviors depending on the side on which you are erring.

```r
myvec[0]
```

```
character(0)
```

```r
myvec[5]
```

```
[1] NA
```

### Subsetting vectors

You can also use vectors to index other vectors! The indexing vector can have duplicates and indices in arbitrary order.

```r
myvec <- c("these", "are", "some", "values")

myvec[c(1,4,2,4)]
```

```
[1] "these"  "values" "are"    "values"
```

Using negative indices results in a *copy* of the original vector *without* the specified indices[4].

[4] This is very different from Python, where using negative indices gives you the elements *from the end* of the list.

```r
myvec[-2]
```

```
[1] "these"  "some"   "values"
```

We can of course use vectors of negative indexes

```r
myvec[c(-1, -2)]
```

```
[1] "some"   "values"
```

We can use boolean vectors to retain only the entries corresponding to `TRUE`. Given the following sequence:

```r
myvec <- 1:10
```

We can select all the *even* values as follows:

```r
myvec[myvec %% 2 == 0]
```

```
[1]  2  4  6  8 10
```

#### Heterogeneous collections

A `list` allows to store elements of different type in the same collection, without coercion.

```r
my_list <- list(
  3.14, "c", 3L, TRUE
)
typeof(my_list[1])
```

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

The output of the above code might be a bit confusin, since we could expect a `double` type.

If you want to get `atomic` values, you have to use `[[` to index.

```r
typeof(my_list[[1]])
```

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

```r
typeof(my_list[[2]])
```

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

Lists can be named and nested. For all intents and purposes this is equivalent to Python's dictionaries.

```r
my_named_list <- list(
  pi = 3.14,
  name = "Listy List",
  geo = list(
    city = "Padova",
    country = "Italy"
  )
)
```

To access elements, either use a chain of `[[`

```r
my_named_list[["geo"]][["city"]]
```

```
[1] "Padova"
```

or use the `$` operator

```r
my_named_list$geo$city
```

```
[1] "Padova"
```

With the `str` function you can look at the structure of nested lists.

```r
str(my_named_list)
```

```
List of 3
 $ pi  : num 3.14
 $ name: chr "Listy List"
 $ geo :List of 2
  ..$ city   : chr "Padova"
  ..$ country: chr "Italy"
```

### Control flow: `if`

```
if (condition) {
  # Do something if condition holds
} else if (second condition) {
  # Otherwise, do something else if the second condition holds
} else {
  # If non of the previous holds, do this
}
```

For example, do different things depending on the type of a vector

```
my_vec <- c(1.0, 3.14, 5.42)

if (is.numeric(my_vec)) {
  mean(my_vec)
} else {
  # Signal an error and stop execution
  stop("We are expecting a numeric vector!")
}
```

```
[1] 3.186667
```

### Control flow: `for` loops

```
for (iteration specification) {
  # Do something for each iteration
}
```

We will use the following data as examples.

```
loop_data <- list(
  a = rnorm(10),
  b = runif(10),
  c = rexp(10),
  d = rcauchy(10)
)
str(loop_data)
```

```
List of 4
 $ a: num [1:10] -1.207 0.277 1.084 -2.346 0.429 ...
 $ b: num [1:10] 0.317 0.303 0.159 0.04 0.219 ...
 $ c: num [1:10] 0.877 0.0146 1.8351 0.5193 1.9963 ...
 $ d: num [1:10] -159.354 -1.608 21.193 0.963 -0.907 ...
```

We want to compute the mean of each of `a`, `b`, `c` and `d` in `loop_data`. A straighforward approach would be

```r
data_means <- list(
  a = mean(loop_data$a),
  b = mean(loop_data$b),
  c = mean(loop_data$c),
  d = mean(loop_data$d)
)
str(data_means)
```

```
List of 4
 $ a: num -0.383
 $ b: num 0.417
 $ c: num 0.855
 $ d: num -20.9
```

This approach has two issues

- Much repetition
- We must modify the code if we ever extend the list

We can do better with a `for` loop

```r
data_means <- list()
for (i in 1:length(loop_data)) {
  data_means <- c(
    data_means,
    mean(loop_data[[i]])
  )
}

str(data_means)
```

```
List of 4
 $ : num -0.383
 $ : num 0.417
 $ : num 0.855
 $ : num -20.9
```

Note, however, that we lost the *naming* of the vector along the way. To fix this, we can do the following.

```r
data_means <- list()
for (name in names(loop_data)) {
  data_means[name] = mean(loop_data[[name]])
}

str(data_means)
```

```
List of 4
 $ a: num -0.383
 $ b: num 0.417
 $ c: num 0.855
 $ d: num -20.9
```

### Functions

Whenever you find yourself copy-pasting the code, create a function instead!

1. The name of the function serves to describe its purpose
2. Maintenance is easier: you only need to update code in one place
3. You don't make silly copy-paste errors

In R a function call has the following shape

```
fn_name(<value1>,
        argument2 = <value2>)
```

And the definition of a function looks like the following. Note that to give a name to the function we just assign it to a variable.

```
my_func <- function(arg1, arg2, named_arg3 = 42) {
  # Do things with arguments
  # The last statement is the return value
  # you can also use the explicit `return(value)` to do early returns
}
```

Let's make an example. Consider the following data.

```
my_list <- list(
  a = rnorm(5),
  b = rcauchy(5),
  c = runif(5),
  d = rexp(5)
)
str(my_list)
```

```
List of 4
 $ a: num [1:5] 0.00986 0.67827 1.02956 -1.72953 -2.20435
 $ b: num [1:5] -1.319 1.453 -37.231 0.164 -4.862
 $ c: num [1:5] 0.1215 0.8928 0.0146 0.7831 0.09
 $ d: num [1:5] 0.0384 1.2302 2.2003 0.9757 0.337
```

we want to *rescale* all the values so that they lie in the range `0` to `1`. Let's first see how to do it on `my_list$a`:

```
maxval <- max(my_list$a)
minval <- min(my_list$a)

(my_list$a - minval) / (maxval - minval)
```

```
[1] 0.6846843 0.8913725 1.0000000 0.1468252 0.0000000
```

Now, instead of copying and pasting the code for all the entries in `my_list`, we define a function `rescale01`

```
rescale01 <- function(values) {
  maxval <- max(values)
  minval <- min(values)
  (values - minval) / (maxval - minval)
}
```

and then we can invoke it, maybe in a loop

```
output <- list()
for (nm in names(my_list)) {
  output[[nm]] <- rescale01(my_list[[nm]])
}
str(output)
```

```
List of 4
 $ a: num [1:5] 0.685 0.891 1 0.147 0
 $ b: num [1:5] 0.928 1 0 0.967 0.837
 $ c: num [1:5] 0.1217 1 0 0.8751 0.0858
 $ d: num [1:5] 0 0.551 1 0.434 0.138
```

You can write functions that accept a variable number of arguments using the `...` syntax:

```
with_varargs <- function(...) {
  # The following line stores the additional arguments in a list,
  # for convenient access. Additional arguments can even be named
  args <- list(...)

  return(str(args))
}
```

```
with_varargs(
  "hello",     # This is a positional argument
  b = 42,      # This is an additional argument that will go in the args list
  a = "world"  # And additional arguments can also be named
)
```

```
List of 3
 $  : chr "hello"
 $ b: num 42
 $ a: chr "world"
```

### Libraries

While functions are the basic unit of code reuse, oftentimes they are grouped together in bundles providing related functionality. Such bundles are called *libraries* (or *packages*).

A comprehensive index of R packages is hosted at https://cran.r-project.org.

To install a library you can just use the command

```
install.packages("name_of_the_library")
```

in your R console.

Among all the libraries available for R we are particularily interested in the `tidyverse` which is an opinionated collection of R packages designed for data science. All packages share an underlying design philosophy, grammar, and data structures.

Install the complete tidyverse with:

```r
install.packages("tidyverse")
```

### Using libraries

To use functions from a library you prepend the package name to the function's name

```r
readr::read_csv("file.csv")
```

To bring all the package's functions into scope

```r
library(readr)
read_csv("file.csv")
```

The second option is more convenient, but some names may mask names already in scope

```r
library(dplyr)
```

```

Attaching package: `dplyr`

The following objects are masked
from `package:stats`:

    filter, lag

The following objects are masked
from `package:base`:

    intersect, setdiff, setequal, union
```

In this case the *shadowed* names are still accessible using their *fully qualified* name

```r
stats::filter
base::intersect
```