

Variables and functions

Silvie Cinková

2025-07-24

Table of contents

1	Tabular data	2
2	Observations vs. aggregations (summaries)	3
3	Data structures	4
4	Data types (aka classes)	5
5	Programmatic objects	5
6	Programmatic variables	6
7	Variable assignment statement	6
8	Variable overwriting	7
9	Functions and their arguments	8
10	Documentation in Help	8
11	Libraries (aka packages)	10
12	Function with an argument	10
13	Function with several arguments	11
14	Coercion to string	11
15	No coercion with error	12
16	Write and run your own function	13

17 Vector	13
18 Vectorized functions	14
19 Non-vectorized case	14
20 Step by step (no automation)	15
21 Vector	15
22 Create a vector, combine them	15
23 Programming basics	16
24 Vector class coercion logical to numeric	16
25 Vector class coercion to character	17
26 Programming with vectors	17
27 Vector subsetting by positions	17
28 Vector subsetting with logical operators	18
29 Vector recycling	18
30 Vector recycling	19
31 Vector recycling	19
32 Vector recycling	19
33 Endnotes	20

1 Tabular data

Table 1: Top longevity Europe 2007

country	LifeExpectancy
Iceland	81.757
Switzerland	81.701
Spain	80.941
Sweden	80.884

France	80.657
--------	--------

Before we delve into programming, let us sort out how we use tables to capture data and generate insights from them. This will help us to understand how things are done with R, because R was designed for statistical computation on tables. You can imagine that most things R does to an objects, it makes sure to keep that object ready to include into a table or convert to one.

Table is a central concept in data science. It has **rows** and **columns**. Columns $\approx$ **statistical variables** (not to confuse with programming variables that we will meet soon).

1. How many observations and how many variables can you see in the table in Table 1?
2. How would you characterize each variable in terms of discreteness-continuity and whether it is qualitative/categorical or quantitative?
3. Sketch for yourself a few tables with different sorts of variables, as you could meet them or need to create in real life.

2 Observations vs. aggregations (summaries)

Table 2: Life expectancy data in two countries

country	year	LifeExpectancy
Albania	2007	76.423
Albania	2002	75.651
Albania	1997	72.950
Denmark	2007	78.332
Denmark	2002	77.180
Denmark	1997	76.110

Table 3: Average life expectancy in Albania and Denmark 1997 - 2007

country	AverageLifeExpectancy
Albania	75.00800
Denmark	77.20733

What are these two tables to do with each other? The one to the left contains observations. The one to the right contains the aggregation of life expectancy for each country across years.

3 Data structures

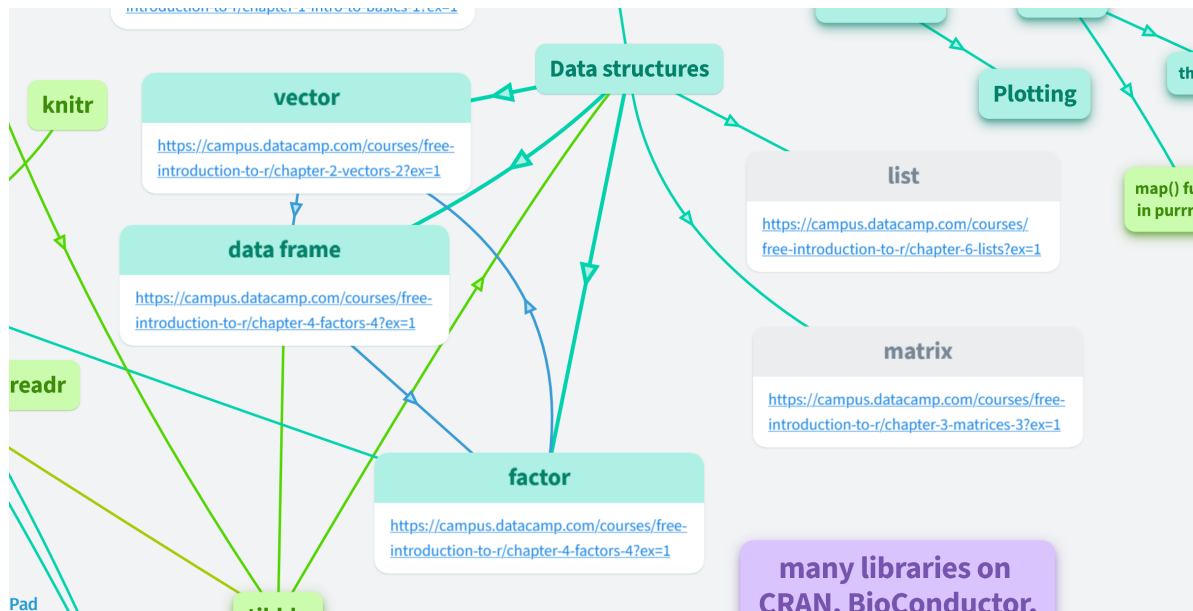


Figure 1: Data Structures in R

Now let us try and trace these concepts in R data structures. The table itself is implemented as **data frame**, or, in the **tidyverse** dialect, **tibble** (a tibble is an enhanced data frame, so let me just stick to *data frame*). Data frames have columns and these columns have names. Each value in a column corresponds to a row. Rows, on the other hand, are not explicitly described in R. All operations on tables are conceived from the perspective of columns. R believes that each column stores one statistical variable and it is ready to aggregate it to a summary statistics with a one-line command.

And now is the right moment to introduce the most basic R data structure, which is the *vector*. A vector is an ordered sequence of *elements*. An element is an individual value, such as a number. In terms of data science, it is the value of a statistical variable in one observation. If you imagine you record the air temperature at noon every day. When you have done this for a week, you have a vector of seven elements. Each element contains the temperature value. To store this as a meaningful information, you will probably keep another vector, in which each element is going to be the date of the measurement. After a week, you are going to have a seven-element vector. You can put them together into a data frame as two columns and give them some column names. Then the first element of the date vector is going to correspond to the first element of the temperature vector, and so on. Vector elements on the same position in the vector are the implicit rows in a data frame.

So when you add a row to a data frame, you actually add one element to each of its column vectors.

Vector and data frame are the most relevant data structures in this course. Let us leave the others (*factor*, *list*, and *matrix*) aside for a while.

4 Data types (aka classes)

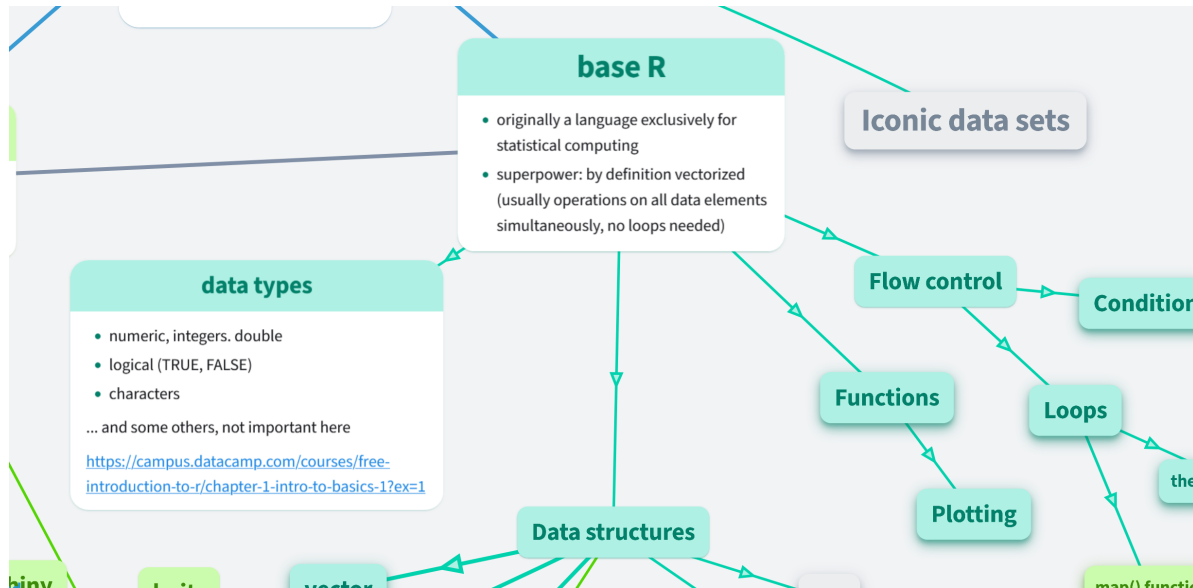


Figure 2: R data types: characters, numbers, TRUE/FALSE (aka logical/Boolean)

Recall what you just rehearsed about statistical variables: categorical or quantitative. How absurd would you find a statistical record that would mix numerical values with verbal descriptors in one column, such as *20*, *25*, and *around twenty* or *forgot to measure it today*! So R simply disallows such a mix in one vector. A vector can contain numbers or characters/strings or Boolean values, but never in combination.

i A terminological note

When a value consists of letters and numbers meant as symbols, it is called a **string**. A string consists of **characters**. Sometimes we use *string* and *character* interchangeably, but usually we refer to a value consisting of one character as a *character* and when it is longer, it is a *string*.

5 Programmatic objects

- **object** $\approx$ anything you enter to the computer without an error message coming

```
3 + 2
```

```
[1] 5
```

The resulting object vanished as soon as it appeared on the screen.

6 Programmatic variables

- **variable** $\approx$ a labeled box in computer memory to store an object

```
my_calculation <- 3 + 2
```

Nothing appeared, but the object exists in the variable.

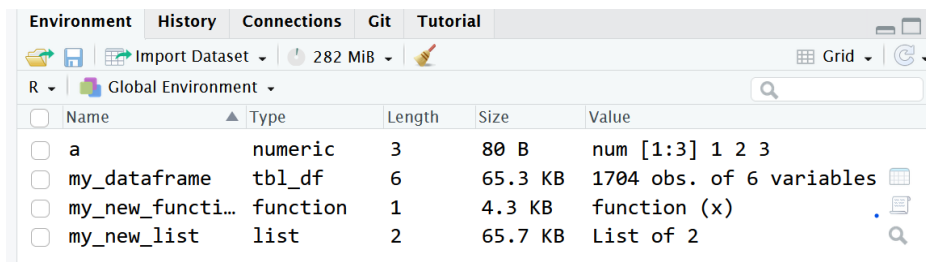
```
my_calculation
```

```
[1] 5
```

7 Variable assignment statement

```
variable_name <- object
```

- variable names must not
 - start with a digit
 - contain any characters but ASCII letters, numbers, and `_` (underscore)
 - coincide with a few reserved words in R (e.g. `for` or `in`)



The screenshot shows the R Environment tab with a table of variables. The table has columns for Name, Type, Length, Size, and Value. The variables listed are 'a' (numeric), 'my_dataframe' (tbl_df), 'my_new_functi...' (function), and 'my_new_list' (list).

☐	Name	Type	Length	Size	Value
☐	a	numeric	3	80 B	num [1:3] 1 2 3
☐	my_dataframe	tbl_df	6	65.3 KB	1704 obs. of 6 variables
☐	my_new_functi...	function	1	4.3 KB	function (x)
☐	my_new_list	list	2	65.7 KB	List of 2

Figure 3: Each variable appears in the Environment tab

The assignment operator consists of two characters: “smaller than” (<) and dash (-) next to each other. You must keep a space before and after this operator. When you assign an object to a variable, it happens quietly. Nothing gets printed out. To print out what is stored in a variable, type its name and run the line.

8 Variable overwriting

- create a different variable for each step in your script

```
a <- 1
b <- a + 2
b
```

```
[1] 3
```

- overwrite the variable

```
a <- 1
a <- a + 2
a
```

```
[1] 3
```

You must save each step in your script to a variable. When you write this:

```
a <- 3
a + 2
```

```
[1] 5
```

```
a
```

```
[1] 3
```

The result (5) has not be stored in **a**.

9 Functions and their arguments

- Functions $\approx$ verbs
- “argument structure”/“valency”
 - obligatory vs. free vs. unacceptable (no slot)
 - collocability
 - * argument does not fit
 - * argument causes a meaning shift

Functions are very similar to content verbs in natural languages. Like verbs, they have a meaning on their own and individual argument structure; that is, they open slots for diverse clause elements.

Argument structure is a syntactic property. Virtually all verbs open a slot for subject. Transitive verbs open another slot for the direct object. These are then obligatory. When any of them remains unfilled, odds are that the whole clause becomes non-grammatical or at least confusing.

Collocability primarily concerns meaning. Many verbs require their arguments to belong to a specific semantic class; e.g. the direct object of *read* would be a textual medium. You can often use the verb metaphorically, such as *read someone's mood* or *read the weather*. Some of these metaphors are so widely used that a dictionary may even capture them as separate senses), but some just would not work. For instance *read a mushroom* would require an explanation.

Each function in R that you load from CRAN has a user documentation. Sometimes it is extremely technical to read, but at least it exists.

10 Documentation in Help

This help page in the R manual describes four similar functions in one bulk. Functions come in packages/libraries. These functions are part of the **base** library, which downloads with every installation of R. You see the library name in the curled braces above.

Normally, each function gets its own documentation page, but such a bulk documentation for a set of related functions is also quite common. All these functions take a character string and either modify its letter case or replace a substring (i.e. piece) with another substring.

Each documentation page has the following sections:

- Description (one line)
- Usage (all arguments listed, with their default values if applicable)

Character Translation and Case Folding

Description

Translate characters in character vectors, in particular from upper to lower case or vice versa.

Usage

```
chartr(old, new, x)
tolower(x)
toupper(x)
casefold(x, upper = FALSE)
```

Arguments

- x** a character vector, or an object that can be coerced to character by [as.character](#).
- old** a character string specifying the characters to be translated. If a character vector of length 2 or more is supplied, the first element is used with a warning.
- new** a character string specifying the translations. If a character vector of length 2 or more is supplied, the first element is used with a warning.
- upper** logical: translate to upper or lower case?

Figure 4: Help - functions to replace strings

- When you use the function, you do not need to list the arguments that have default values, unless you want to override the default value with your own choice.
- Arguments (each argument described)
- Details (a more description)
- Value (what output you get from the function)
- Examples (thankfully, you can run a few examples and observe the function's behavior in the Help pane.)

11 Libraries (aka packages)

- additional functions and data sets
- first install (**Packages** tab)
- load when you need a function from there
 - `library(thelibrary)` without quotes
- when functions from different packages happen to have the same name
 - R warns when it finds such a pair across your loaded libraries.
 - Use these functions with package name like this: `library::function()`

12 Function with an argument

```
toupper(x = "hello")
```

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

This function is called `toupper` and requires one argument that it calls `x`. This argument is supposed to be a string. The function returns the same string, only with all letters converted to upper case if they were originally lower case.

Each function has a name and requires some arguments in parentheses. Their number, names and accepted data class are a decision of the programmer who wrote that function.

Functions in R have been written by many different authors over several decades. You must not expect that they are all written in the same way. Sometimes the value that you input is

called `x` like here, but that is a fully arbitrary decision of the author of that function. Other functions can call it differently. To make sure to use a new function correctly, you must read its documentation in the **Help** tab.

13 Function with several arguments

```
chartr(old = "a", new = "A", x = "banana")
```

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

```
chartr(new = "A", x = "banana", old = "a")
```

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

```
chartr("a", "A", "banana")
```

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

When you write out the names of the arguments, you can list them in any order. When you omit the names, R will believe they are ordered like in the documentation.

14 Coercion to string

```
toupper(3)
```

```
[1] "3"
```

```
toupper(TRUE)
```

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

In this example, I fed this function a number and a Boolean value instead of a string. The function accepts either, but outputs them as strings, as you can tell from the quotes that surround the printed output. So, behind the scenes, this function converts the input to a character. This behavior is called **coercion** and is typical of R. Rather than throwing an error, a function may *coerce* a wrong input class to becoming the required class - but it is not always the case. Again, it depends on how the R developer in charge wrote it.

15 No coercion with error

```
chartr(old = "mo", new = "fa", x = "mother") # strings
```

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

```
chartr(old = "5", new = "0", x = "5toFour") # digits as strings
```

```
[1] "0toFour"
```

```
chartr(old = "5", new = 0, x = "5toFour" ) # new won't work
```

```
Error in chartr(old = "5", new = 0, x = "5toFour"): invalid 'new' argument
```

The `chartr` function takes three arguments: `x` is the input string, `old` is a string sequence in that string you want to find, and `new` is what you want to replace the old with. Both must have the same number of characters. In the first example, `old` and `new` are unarguably strings. In the second example, we served the digits as strings by enclosing them in the quotes, and it worked, too. However, in the third example, no coercion took place, and the function threw an error when the `new` argument contained a digit. And this is what you met in two functions that belong to the same package, do related things and have even been described bulk-wise in one `Help` entry. Were you expecting the digit to be coerced to a string like with `tolower`? I believe your expectation was fully legitimate.

16 Write and run your own function

```
my_function <- function(my_1_arg,
                        my_2_arg) {
  my_argument <- paste(my_1_arg, # existing R function
                      my_2_arg,
                      sep = "--")
  toupper(my_argument)
}
# run it like so:
my_function(my_1_arg = "Hello",
           my_2_arg = "again")
```

```
[1] "HELLO--AGAIN"
```

Writing your own function is as easy as this:

1. Make a variable assignment (`<-`) to a new variable name. This is going to be the name of your function.
2. Type `function`.
3. Add a pair of parentheses and list arguments that your function will require.
4. Write the function body between a pair of curly braces.

Then run this code. The name of the function appears in the `GlobalEnvironment` in the `Environment` pane.

Finally, call your function and give it the required arguments.

17 Vector

- `c()` function to concatenate / combine values to a vector

```
my_vector <- c(200, 0.3, 8)
my_vector
```

```
[1] 200.0    0.3    8.0
```

```
str(my_vector) # shows object's structure: numeric vector of 3 elements
```

```
num [1:3] 200 0.3 8
```

18 Vectorized functions

```
nice_strings <- c("HOME", "EYE", "SET") # create vector
result_vector <- tolower(nice_strings) # process with tolower
str(result_vector)
```

```
chr [1:3] "home" "eye" "set"
```

```
chartr(new = "e", old = "E", x = result_vector)
```

```
[1] "home" "eye" "set"
```

What you see in the slide: You give `tolower` a character vector and it returns the same vector but elements converted to lower case.

Most R functions are **vectorized** like `tolower`. That means that you give the function a vector as argument, and the function processes all elements of the vector simultaneously, also returning the result in a vector.

When a function is not vectorized, it either takes a vector and returns one single value, or it requires a single value at input. When you need to process all elements of a vector with such a function, you must run the function on each vector element separately, using a `loop` or an `apply` function or a function from the `purrr` package. (Let us ignore those for the moment.)

19 Non-vectorized case

- `chartr` cannot do a vector of replacements:

```
chartr(old = c("a", "b"), new = c("A", "B"), x = "banana")
```

```
Warning in chartr(old = c("a", "b"), new = c("A", "B"), x = "banana"): argument
'old' has length > 1 and only the first element will be used
```

```
Warning in chartr(old = c("a", "b"), new = c("A", "B"), x = "banana"): argument
'new' has length > 1 and only the first element will be used
```

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

20 Step by step (no automation)

```
banana_1 <- chartr(old = "a", new = "A", x = "banana")  
banana_1
```

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

```
banana_2 <- chartr(old = "b", new = "B", x = banana_1)  
banana_2
```

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

This would be a perfect use case for a loop, but there are still more basic things to mention.

21 Vector

- Class
 - numeric, character, logical
- Length (how many elements)
- Elements can have names (named vector).
- Order of elements matters.

`class` says whether R should treat the elements as numbers, or as characters or Boolean values. All elements of one vector are always of the same class. You will see soon what happens when you mix elements from different classes. The `numeric` class has two subclasses: `integer` and `double`. R automatically scrutinizes the elements and assigns the class automatically. You do not need to worry about a numeric class being silently changed into `integer` or `double`.

22 Create a vector, combine them

```
(a <- c(23:27)) # colon generates incremental sequence
```

```
[1] 23 24 25 26 27
```

```
(b <- c(a, FALSE, TRUE))
```

```
[1] 23 24 25 26 27 0 1
```

23 Programming basics

Put each command on one line.

When you enclose an assignment statement in parentheses, the variable will print out.

What you see in the second chunk is called **class coercion**.

24 Vector class coercion logical to numeric

- logical + numeric = numeric
- boolean values translate to 0 and 1

```
boolean_vec <- c(TRUE, TRUE, FALSE) # logical
numeric_vec <- c(3, 2, 6) # numeric
c(boolean_vec, numeric_vec)
```

```
[1] 1 1 0 3 2 6
```

```
class(c(boolean_vec, numeric_vec))
```

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

You have to watch out for the class being changed from any kind of **numeric** to **character** or from **logical** to **numeric** or **character**. Whenever you wonder why your numbers result in a character vector, you must check each element for non-numeric characters. The most common causes of a sequence of apparent numbers being resolved as characters are a space after the digit, or the decimal sign not corresponding to the locale of your system (by default, R is US-centric, so it uses decimal points, not commas). This is extremely important because wrong class on input to a function may either cause error, or worse, you get something different from what you expected. Most typically, if you call the function `sort` on numbers, they will get sorted in the linear order, but when this vector silently converts to characters, they will be sorted alphabetically. In the former case, you will get 1,2,3...11. In the latter case, you will get 1, 11, 2, 3...

25 Vector class coercion to character

- anything + character = character
- even digits can be characters

```
mix_vec <- c(boolean_vec,  
             numeric_vec,  
             "June 8", "3 ")  
class(mix_vec)
```

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

26 Programming with vectors

- select elements of a vector
 - by their position
 - by some condition

This will take you away from data frames for a while, but it is the very fundamentals of the R language. You would really struggle with the data frames if you had no idea of how to deal with vectors. Selecting elements from a vector is a training for filtering rows of a data frame, even in *tidyverse* (let alone with base R, should you ever be forced to do it!)

27 Vector subsetting by positions

```
vec <- c("John", "Mary", "Paul")  
vec[1]
```

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

```
vec[2:3]
```

```
[1] "Mary" "Paul"
```

```
vec[c(1,3)]
```

```
[1] "John" "Paul"
```

You can pick elements from the vector with an operation called **subsetting**. The subsetting operator is a pair of square brackets with the indices (numbers) of the positions of the elements you want to pick up.

28 Vector subsetting with logical operators

```
vec <- c(1, 20, 3, 4)
vec[vec < 20]
```

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

```
vec[vec < 20 & vec > 1] # and
```

```
[1] 3 4
```

```
vec[vec > 15 | vec < 3] # or
```

```
[1] 1 20
```

29 Vector recycling

- many functions proceed element by element
- nothing gets recycled with equally long vectors

```
c(2,4,6,8)/c(2,2,2,2)
```

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

```
c(1000, 100, 10) / c(100, 10, 1)
```

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

30 Vector recycling

- The second vector contains just one value and that must serve each element of the first vector
- 2 gets *recycled*

```
c(2,4,6,8) / 2
```

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

31 Vector recycling

- the second vector gets recycled once
- each of its element must serve twice
- R believes you want it this way

```
c(2,4,6,8) / c(2,1)
```

```
[1] 1 4 3 8
```

32 Vector recycling

- Did you really want this? Warning.

```
c(10, 10, 10) * c(1,2)
```

Warning in `c(10, 10, 10) * c(1, 2)`: longer object length is not a multiple of shorter object length

```
[1] 10 20 10
```

33 Endnotes

1: Variables = columns (two), observations = rows

2: Qualitative/categorical variables are always discrete: When the values are names of countries like in the example, you cannot have a value that would lie, e.g., between Denmark and Sri Lanka. Life expectancy is a quantitative value and it is continuous. When you see neighboring values, it is very well possible that another country's life expectancy would still lie between. When you disregard rounding, you could see extremely tiny differences, for instance five seconds or so... On the other hand, year is usually interpreted as a discrete variable, although time is unarguably a continuous concept.