Fundamental object types in R III: Factors and data frames

Claudius Gräbner-Radkowitsch

2022-03-02

Contents

1 Optional background info: attributes and classes

In the previous post you learned about the most important fundamental data types in R. The types we will learn about below and not less important, but less fundamental. This means they are built by taking one of the base types we encountered before, and 'adding some features'. These features change the behavior of the type, e.g. how it is printed or how it is affected by certain function calls, but also what kind of operations it allows.[1]

This process of 'adding features' is usually done by adding 'attributes' to an object. In principle, you can add attributes to any objects without much effect by using the `attr()` function:

```
x <- 2.0
attr(x, "Mood") <- "Haha!"
```

To retrieve attributes use `attributes()`:

```
attributes(x)
```

```
## $Mood
## [1] "Haha!"
```

Sometimes, adding attributes of a particular name have more relevant implications. One useful way to use attributes, for instance, is to name the single elements of vectors, something that changes the way the objects are printed and something that we already discussed in the context of lists:

```
y <- c(1, 2, 3)
attr(y, "names") <- c("First", "Second", "Third")
y
```

```
##  First Second  Third
##      1      2      3
```

[1] In fact, some new types allow you to do *less* than the original type, i.e. here the new features are restrictions. The `tibble` we will encounter below is such an example.

```
attributes(y)
```

```
## $names
## [1] "First"  "Second" "Third"
```

But things become really interesting if you add an attribute called `class` since this really transforms the data types into a new, less fundamental type. In fact, this is how the types we discuss below, are created: smart people added, among other things, a class attribute to a more fundamental data type (`integer` in the case of `factor`s and `list` in the case of `data.frame`s). The art of writing new classes is part of object oriented programming, an advanced concept that we do not cover in this course (and, to be honest, one of the areas where R is not particularly well designed).

One implication of this 'less fundamental' nature of the objects we encounter below is that `typeof()` usually returns the base type. For instance, below we will learn about the `factor`, a type that is built upon `integer`. If we call `typeof()` on a `factor`, it will return the fundamental type, i.e. `factor`:

```
xx <- factor(c(1,2))
typeof(xx)
```

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

Fortunately, the standard test functions (`is.*()`) usually work, so you can use `is.factor()`:

```
is.factor(xx)
```

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

Alternatively, you can always inspect that attributes of the object to find out about its class:

```
attributes(xx)
```

```
## $levels
## [1] "1" "2"
##
## $class
## [1] "factor"
```

All this can be confusing at first, so it is important to keep this in mind. Once you wrapped your head upon this, many confusing behaviors suddenly start to make sense, e.g. that mutating factors within a `data.frame` can result in whole numbers, a phenomenon we will discuss in the context of data wrangling later.

2 Factors

We usually use factors to represent ordinal or categorical data. At first sight a factor is an atomic vector that can only take a pre-specified number of values, so called *levels*. To create a factor we use the function - surprise - `factor()`:

```
x <- c("Female", "Male", "Female")
x <- factor(c("Female", "Male", "Female"))
x
```

```
## [1] Female Male   Female
## Levels: Female Male
```

If we want to define levels that do not yet have any instances, we can do this with the optional argument `levels`:

```
x <- c("Female", "Male", "Female")
x <- factor(c("Female", "Male", "Female"),
```

```
            levels=c("Diverse","Female", "Male"))
x
```

```
## [1] Female Male   Female
## Levels: Diverse Female Male
```

If we use the argument levels and also try to add values that are not included in the levels, these are set to `NA`:

```
x <- c("Female", "Male", "Female")
x <- factor(c("Female", "Male", "Female", "Diverse"),
            levels=c("Female", "Male"))
x
```

```
## [1] Female Male   Female <NA>
## Levels: Female Male
```

Usually, the sequence in which we mention levels does not matter. In the case of ordinal data, however, the sequence becomes important. To consider it, we must use the argument `ordered`:

```
x <- c("High", "High", "Low", "High")
x <- factor(x,
            levels = c("Low", "Mid", "High"),
            ordered = TRUE)
x
```

```
## [1] High High Low  High
## Levels: Low < Mid < High
```

Frequently, the elements of factors are words, i.e. objects of the type `character`. From a technical point of view, however, factors are stored as `integers`: in order to save memory space, each level is assigned a whole number in the computer memory, which is then mapped to the actual value. Especially when the elements consist of large numbers or long words, this helps saving memory space since these expressions only have to be stored once, and each element of the factor is only a simple number. Therefore, `typeof()` also returns `integer` for factors:

```
x <- factor(c("Female", "Male", "Female"),
            levels=c("Male", "Female", "Diverse"))
typeof(x)
```

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

To check whether an object is a factor we use the function `is.factor()`:

```
is.factor(x)
```

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

Not all operations that are defined for `integer`s, however, also work for `factor`s:

```
x[1] + x[2]
```

```
## Warning in Ops.factor(x[1], x[2]): '+' not meaningful for factors
```

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

At the same time, there are some useful things we can do with factors. For instance, the function table gives us the absolute frequencies of the factor elements, a task that is very common for categorical data:

```
table(x)
```

```
## x
##    Male  Female Diverse
```

```
##       1       2       0
```

3 Data Frames

The `data.frame` is a special type of list. It is among the most widespread data types used in data analysis. In contrast to a normal list, all elements of a `data.frame` must have the same length. This means that you can think of a `data.frame` as a list arranged as a rectangle and represented as a table. The headings of the tables then correspond to the names of the vector elements of the list.

Because of the close relationship, we can create a `data.frame` directly from a list by using the function `as.data.frame()`:

```
l_3 <- list(
  "a" = 1:3,
  "b" = 4:6,
  "c" = 7:9
)
df_3 <- as.data.frame(l_3)
```

Here we might think of the names of the three vectors, `a`, `b` and `c` as the headings of a table (representing, often, variable names), and the content of the vectors as the cell entries of the table:

```
df_3
```

```
##   a b c
## 1 1 4 7
## 2 2 5 8
## 3 3 6 9
```

The relation to `lists` becomes obvious if we call `typeof()`, which returns the underlying data type:

```
typeof(df_3)
```

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

To test whether an object is a `data.frame` we use `is.data.frame`:

```
is.data.frame(df_3)
```

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

```
is.data.frame(l_3)
```

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

The difference to a list becomes particularly obvious in the different printing behavior of the two:

```
l_3
```

```
## $a
## [1] 1 2 3
##
## $b
## [1] 4 5 6
##
## $c
## [1] 7 8 9
```

```
df_3
```

```
##   a b c
```

```
## 1 1 4 7
## 2 2 5 8
## 3 3 6 9
```

A more direct way to create a `data.frame` is to use the function `data.frame()`:

```
df_4 <- data.frame(
  "gender" = c(rep("male", 3), rep("female", 2)),
  "height" = c(189, 175, 180, 166, 150)
)
df_4
```

```
##   gender height
## 1   male    189
## 2   male    175
## 3   male    180
## 4 female    166
## 5 female    150
```

To extract single elements, columns or rows of a `data.frame` we can use `[` and `[[` in a similar way we used it for lists, just keeping in mind that we now have two dimensions to deal with, instead of only one in the case of lists.

To subset columns we can call them by name:

```
df_4["gender"]
```

```
##   gender
## 1   male
## 2   male
## 3   male
## 4 female
## 5 female
```

The result is another `data.frame`:

```
is.data.frame(df_4["gender"])
```

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

If we want to extract the underlying atomic vector we must use `[[`:

```
df_4[["gender"]]
```

```
## [1] "male"   "male"   "male"   "female" "female"
```

This is also helpful to inspect the underlying data type:

```
typeof(df_4[["gender"]])
```

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

Data frames can become very large, so its often useful to get a first overview about what the `data.frame` contains. The functions `names()`, `dplyr::glimpse()`, `head()`, and `View()` are useful in this context.

The function `names()` returns a vector with all the column names:

```
names(df_4)
```

```
## [1] "gender" "height"
```

> **Hint:** For large `data.frames` it can be useful to wrap `names()` into `sort()`, which sorts the resulting vector in alphabetical order.

Head only prints the first `n` rows of a `data.frame`. By default `n=5`, but you can set it explicitly:

```
head(df_4, n = 2)
```

```
##   gender height
## 1   male    189
## 2   male    175
```

The package `dplyr` proves the useful function `dplyr::glimpse()`, which gives a general overview about all columns and their types:

```
dplyr::glimpse(df_4)
```

Finally, if you work within R Studio, you can use `View()` to get a Excel-like representation of your data.

As you might have guessed by now, `data.frame`s are the classic object to represent read-in data. If, for instance, you download data on GDP in Germany from the Internet and then import this data into R, it usually gets represented as a `data.frame`. This representation then allows direct analysis and manipulation of the data.

At the same time, `data.frame`s are a very old object type, the behavior of which sometimes feels a bit outdated. Because of this, some authors created a new data type, which is built upon the `data.frame`, but behaves slightly different: the `tibble`. The tibble is made available via the package `tibble`, so to use tibbles, this packages has to be loaded first.

You can read more about tibbles in chapter 10 of R for Data Science, i.e. here (to avoid confusion: in his book Hadley always attaches the package `tidyverse` at the beginning of the chapter. This includes attaching the package `tibble`.)

> **Digression: Dialects of R** As with natural language, programming languages feature different dialects, i.e. different ways of doing the same actions. In R this becomes particularly obvious in the context of dealing with data. Here, the different dialects show themselfes in different object types used for the data you are working with. To the best of my knowedge, there are three main dialects, each associated with a different focus on data representation. The *base* dialect represents data as `data.frame`s and it is the original approach envisioned by the R developers. The *tidyverse* dialect represents data as `tibble`s and is built around a number of different packages summarized under the name tidyverse. It is aimed to develop an set of packages that allow you to do all tasks commonly associated with data science within a consistent syntax and design philosophy. The *data.table* dialect represents data as `data.table`s, which are provided by the package data.table. It is designed to work with large data sets and is by far the fastest and computationally most efficient approach to data processing. In this course we will follow the `tidyverse` dialect because it is particularly easy to learn, widespread, and usually a good default option. Once you mastered it and you want to work in the field of big data, you should then consider learning the data table dialect as well. Right now, the most important thing is consistency: all the dialects help you to achieve almost any aim you will ever have in the context of data science. But things get very confusing if you mix different dialects in practive. Their functions are designed to work well with the other elements of the respective dialect environment, but things can become confusing if you mix data types and functions from, e.g., the tidyverse and data.table dialect. If you want to read more about the controversies associated with the different dialects you might start, e.g., here (altough I do not share the opinion of the author regarding teachability).

Transforming `data.frame`s and `tibble`s is one of the central tasks when doing data science, and there are different approaches to it in R. Usually, taking your raw data and turinging it into a well formatted `data.frame` is not straightforward and usually takes at least as much time as the final analysis of the data. Thus, it is one of the central course objectives to equip you with the tools to facilitate this often underrated and underestimated task. But this will be the main subject of the sessions on data wrangling.

4 Digression: Matrices in R

Another data type that is used frequently in many circumstances, especially statistics and engineering, are matrices. These are two-dimensional objects with rows and columns, each of which are atomic vectors.[2] We do not deal with matrices explicitly to a large degree, but knowing about how they work often turns out useful. Therefore, I provide a quick overview below, but since matrices are not a central subject of this course, I leave you with some links to further information.

Matrices are created with the function `matrix()`. This function takes as its first argument the elements of the matrix and then the specification of the number of rows (`nrow`) and/or the number of columns (`ncol`):

```
m_1 <- matrix(11:20, nrow = 5)
m_1
```

```
##      [,1] [,2]
## [1,]   11   16
## [2,]   12   17
## [3,]   13   18
## [4,]   14   19
## [5,]   15   20
```

We can extract and/or substitute the rows and columns as well as individual values as follows:

```
m_1[,1] # First column
```

```
## [1] 11 12 13 14 15
```

```
m_1[1,] # First row
```

```
## [1] 11 16
```

```
m_1[2,2] # Element [2,2]
```

```
## [1] 17
```

> **Hint:** Matrices are built upon atomic vector, which is why `typeof()` always returns the type of the atomic vectors making up the elements:

```
typeof(m_1)
```

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

> As with factors and data frames this has to do with the particular class attributes of matrices, which, however is *implicit*:[3]

```
class(m_1)
```

```
## [1] "matrix" "array"
```

```
attributes(m_1)
```

```
## $dim
## [1] 5 2
```

> To test whether an object is a matrix use `is.matrix()`:

```
is.matrix(m_1)
```

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

[2] If you read the section on attributes above: matrices are objects that were given a new attribute `dim`.

[3] This is another piece of evidence for the very confusing class concept and object-orientated style of R. See this chapter for more details, in case you are interested.

```
is.matrix(2.0)
```

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

The most important thing to learn about matrices is matrix algebra. You can find many good introductions in the web, e.g. here or here and, in German, here.