Handling special data in R

Antonio Gasparrini

01 November 2023

Contents

This session presents data management topics involving special data types and computational methods in R. Some of the required tasks to handle these data make use of specific features of the R language, with related symbols and syntax. The first section illustrates the definition and handling of *missing* data, defined in R using the letters NA (not available) and similar. The second section describes special computations needed to deal with *strings* (or character variables), using functions that compute on text. The third and last section introduces methods for dealing with *dates* (and more generally date/time data), which require specialised formatting and computations.

Working with missing data

The presence of missing observations for some variables is a frequent problem in data analysis. As any other major statistical software, R treats missing values in special ways. For insance, R uses the special constant `NA` to refer to observations which are *not available*. Hence, `NA` is not a value, but a marker for a quantity that is not known. This is a subtle difference, which affects the computations involving objects with missing values.

We demonstrate some features using examples involving the dataset BIRTHS from the package EPI. This package is neither a standard nor a recommended one, and therefore it must be installed (the first time) and then loaded. In addition, the data frame BIRTHS should be loaded as well into the session:

```
library(Epi)
data(births)
```

Let's have a look at the first records:

```
head(births, 3)
```

```
##   id bweight lowbw gestwks preterm matage hyp sex
## 1  1    2974     0   38.52       0     34   0   2
## 2  2    3270     0      NA      NA     30   0   1
## 3  3    2620     0   38.15       0     35   0   2
```

You can notice that, in the second record (row), the values of `gestwks` and `preterm` , namely the gestational week of the birth and an indicator if this is identified as being pre-term, respectively, are set to missing by coding them as `NA`. (see `help(births)`).

A preliminary task often required in data management is to check which observations of a variable are missing. A straightforward method would be to test which observations are equal to `NA` by using logical expressions with the operator '=='. Let's try out and apply the condition to the first observations of `gestwks`:

```
head(births$gestwks) == NA
```

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

Surprisingly, the result is a vector with all the elements as missing. The reason is that `NA` is an unknown quantity, so also the result of the logical condition cannot be determined. The task is instead completed by using the function `is.na()`:

```
is.na(head(births$gestwks))
```

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

which returns a logical vector with `TRUE` corresponding to missing values in the original variable. Here, correctly, the second observation is identified as missing.

The information stored in the logical vector can be used to exclude the non-missing values from `gestwks` through indexing. For instance (using pipes to simplify the code):

```
births$gestwks |> head(4)
```

```
## [1] 38.52    NA 38.15 39.80
```

```
births$gestwks[!is.na(births$gestwks)] |> head(4)
```

```
## [1] 38.52 38.15 39.80 38.89
```

Similarly, missing values can be inputted in any variable by assigning elements to `NA` in replacement expressions.

Following the same concept about the language definition of `NA`, computation involving missing values in R require special handling. For instance, let's try to compute the average gestational weeks of births:

```
mean(births$gestwks)
```

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

The result is missing, as the average cannot be known if at least one of the elements is not known (*i.e.*, missing) itself. As many other functions applied to produce computations involving the full set of elements in vectors or matrices, the function `mean` has an argument `na.rm` (by default set to `FALSE`) which can be used to remove the missing values without the need to call `is.na()` as above. Let's see:

```
mean(births$gestwks, na.rm=T)
```

```
## [1] 38.72186
```

R also provides general functions for handling missing data in data frames. For instance, the function `na.omit()` can be used to exclude records with missing values from data frames (note: it also works with vectors). An example:

```
na.omit(births) |> head(4)
```

```
##   id bweight lowbw gestwks preterm matage hyp sex
## 1  1    2974     0   38.52       0     34   0   2
## 3  3    2620     0   38.15       0     35   0   2
## 4  4    3751     0   39.80       0     31   0   1
## 5  5    3200     0   38.89       0      33   1   1
```

You can notice that the second record has been removed from `births` (check the row names). Similary, the variable `complete.cases()` returns a logical vector indicating which records are complete, *i.e.* have no missing values (see `help(complete.cases)`).

Computing with strings

Specialised data management procedures are needed to deal with strings, which in R are stored in objects of mode `character`. As a programming language itself, R offers a vast collection of functions and methods for computing with strings, which simplify many of the tasks.

One of the most used functions is `paste()`, which concatenates vectors after converting them to character values. As an example, we can use it to create labels by putting together multiple strings and/or other symbols. An example:

```
(subvec <- paste("Subject", ".", 0, 1:4, sep=""))
```

```
## [1] "Subject.01" "Subject.02" "Subject.03" "Subject.04"
```

The function takes an arbitrary number of arguments, accepting objects which can be converted to character vectors . By default, these vectors are then concatenated element by element, separated by the string specified in the argument `sep` (by default, a blank space). In this case, we concatenate the strings `"Subject"` and `"."` with the number 0 and a vector with a regular sequence. Recycling rules apply, so the first three objects are repeated consistently with the length of the last one.

Another argument of `paste()` not used above is `collapse`, which may concatenate again the final vector with another separator. The default is `NULL`, meaning that this second step is not undertaken. Let's see an example:

```
paste("The subjects are:", paste(subvec, collapse=", and "))
```

```
## [1] "The subjects are: Subject.01, and Subject.02, and Subject.03, and Subject.04"
```

In this expression the function `paste()` is called twice, producing a quite elaborate text with a simple code. The innermost call produces a single string, concatenating the names stored in `subvec` with a comma and the text 'and', while the outermost one concatenates this string with the first part of the sentence. These examples demonstrate the extreme flexibility of the software for producing complex character vectors with short and simple code. See `help(paste)` for further details.

Another useful function for computing with strings is `substr()`, which extracts or replaces part of the strings in character vectors. For instance, let's assume we have a vector of names preceded by a code number, and we want to get rid of the latter and extract the names only. This can be easily accomplished by:

```
(namevec <- c("01 Mark","02 Tim","03 Robert"))
```

```
## [1] "01 Mark"   "02 Tim"    "03 Robert"
```

```
substr(namevec, start=4, stop=100)
```

```
## [1] "Mark"   "Tim"    "Robert"
```

The two arguments `start` and `stop` defines the positions of the characters to be extracted. The latter is inputted here as a very high number, as we are interested in keeping the rest of the name, no matter how long. These arguments also accept vectors, if the starting and stopping points need to be different for each element of the character vector. The function `substr()` can also be used in replacement expressions to substitute part of the strings.

The function `match()` and the related operator '`%in%`' are instead used for matching. Let's see some examples:

```r
match(1:5, 3:6)
```

```
## [1] NA NA  1  2  3
```

```r
1:5 %in% 3:6
```

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

The former returns a vector of the positions of (first) matches of its first argument in its second, while the latter returns a logical vector with `TRUE` if each element of the left-hand side matches any element of the right-hand side (see the related help pages.

Two more functions used for pattern matching and replacement are `gsub()` and `grep()`. The former searches for a given pattern in each element of a character vector, and substitute it with a given string. For example, it is sometimes needed to exclude blank spaces from strings used in statistical software, as they are not always accepted. As an example, let's replace the blank spaces in the vector `namevec` created above with a dot:

```r
gsub(" ", ".", namevec)
```

```
## [1] "01.Mark"   "02.Tim"    "03.Robert"
```

The first two arguments of `gsub()` specify the pattern to be searched and the replacement string, respectively. In contrasts, the function `grep()` returns the index positions of strings in a character vector that includes a certain pattern. For instance, let's check which of the names in `namevec` include the letter 'a':

```r
grep(pattern="a", x=namevec)
```

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

The first argument defines the pattern, while the latter is the character vector where matches are sought. In this example, the result indicates that the first name in `namever` contains an 'a'. Of course, the pattern to be searched could be a longer string. The companion function `grepl()`. return the same information but using a logical vector of `TRUE`/`FALSE`.

Other useful functions, not described in examples here, are `nchar()` and `strsplit()`. The former returns the number of characters for each element in a vector, while the latter splits the elements of a character vector into sub-strings, according to the matches to some pattern. See the related help pages for more specific information.

More complex tasks can be accomplished with *regular expressions*, basically the description of a pattern through the use of basic programming language (see `?'regular expression'`). The complexity of the topic is beyond the aim of these notes, but of course you are free to approach it on your own.

Working with dates

Dates are stored and managed in R using objects of specific classes. The most used class is `Date`, which stores dates as *elapsed* days since an origin, corresponding by default to the 1st of January 1970, with negative values for earlier dates. Dates can be inputted in R using the *coercion* function `as.Date`. Let's see some simple examples:

```r
as.Date(c("2000-01-01","1945-04-25"))
```

```
## [1] "2000-01-01" "1945-04-25"
```

```r
as.Date(1:5, origin="1992-12-28")
```

```
## [1] "1992-12-29" "1992-12-30" "1992-12-31" "1993-01-01" "1993-01-02"
```

The function accepts either character or numeric values. In the former, the character vector is transformed in a `Date` object using the strings as labels. In the latter, the numeric vector is interpreted as the number of days since the `origin`, which in this case must be specified. In both cases, the results are stored as numeric values with labels (similarly to factors).

The format\index{format} of the character vector used to input dates, by default *year-month-day*, can be modified using expressions with symbols identifying specific parts of a date. Some examples:

```
as.Date("27/02/94", format="%d/%m/%y")
```

```
## [1] "1994-02-27"
```

```
as.Date("03 July 1233", format="%d %b %Y")
```

```
## [1] "1233-07-03"
```

The expression in the argument `format` defines how the string must be interpreted. Specifically, the symbol ‘`%d`’ defines the day of the month, ‘`%m`’ or ‘`%b`’ the month (in numeric and character format, respectively), and ‘`%Y`’ or ’`%y'` the year (with and without century). See `help(striptime)` for more information). These symbols can be included in the expression together with separators.

As an example with real data, let’s use the dataset AIRQUALITY from the package DATASET, which stores daily air quality measurements in New York from May to September 1973. Let’s have a look at the data:

```
head(airquality, 3)
```

```
##   Ozone Solar.R Wind Temp Month Day
## 1    41     190  7.4   67     5   1
## 2    36     118  8.0   72     5   2
## 3    12     149 12.6   74     5   3
```

The time information is stored in the variables `Month` and `Day`, with the year left undefined. We can therefore create a `date` variable by:

```
airquality$date <- paste(1973, airquality$Month, airquality$Day) |>
  as.Date(format="%Y %m %d")
head(airquality, 3)
```

```
##   Ozone Solar.R Wind Temp Month Day       date
## 1    41     190  7.4   67     5   1 1973-05-01
## 2    36     118  8.0   72     5   2 1973-05-02
## 3    12     149 12.6   74     5   3 1973-05-03
```

The function `format()` can be used to extract specific information about the date from an existing object of class `Date`. For example:

```
(today <- Sys.Date())
```

```
## [1] "2023-11-01"
```

```
format(today, "%a %d %b %Y")
```

```
## [1] "Wed 01 Nov 2023"
```

```
format(today, "%j")
```

```
## [1] "305"
```

The function `Sys.Date()` returns the current date, while the additional symbols ‘`%a`’ and ‘`%j`’ are used to extract the day of the week and year, respectively. See also the functions `weekdays()` and `months()`. Be aware that the function `format()` returns a character vector, and not an object of class `Date`.

The elapsed format is particularly convenient for producing computations with dates, in particular addition and subtraction:

```
today + 1000
```

```
## [1] "2026-07-28"
```

```
today - as.Date("1900-01-01")
```

```
## Time difference of 45229 days
```

The first command is used to identify the date corresponding to the 1000th day since today, while the latter returns the number of days between today and the beginning of the last century.

Additional classes and functions for dealing with dates can be found in contributed packages. Information on date and time requires more complex objects of classes `POSIXlt` or `POSIXct` (see `help(DateTimeClasses)`). The packages LUBRIDATE and CHRON provide an extensive set of functions for producing elaborate computations with dates.