[DS-04] Dates and strings in R

Miguel-Angel Canela
Associate Professor, IESE Business School

Dates

In computer environments, there are typically two data types for time, called date and datetime. In type **date** we can store dates, that is, year, month and day. Most software applications for data management and analysis can deal with different date formats. The default format in R is `yyyy-mm-dd`. I advise you to use only this, even in Excel, which chooses the date format based on the standard of the actual countries.

There are many R packages specialized in date management, and some of them can be required by packages specific for time series analysis. Top popular are `chron` and `lubridate`. I skip these packages here, using only plain R.

Let me show briefly you how to manage dates in R. First, I enter a date as a string:

```
> d1 = '2019-01-01'
> class(d1)
 [1] "character"
```

The function `as.Date` transforms a string into a date, which is an R class, different from numeric, character and logical. We can manage vectors of dates just as we do with vectors of numbers. A data frame can include date vectors together with numeric and character vectors. `as.Date` is a vectorized function, so we can it create date vectors with it.

```
> d2 = as.Date(d1)
> class(d2)
 [1] "Date"
```

`d1` and `d2` may look the same, but they are not, as the function `class` shows. Under the hood, an object of class `date` is just a number, the number of days since the origin of time, which in R is 1970-01-01 (other software applications may use different origins). We can see that number of days by coercing the date to numeric class.

```
> as.numeric(d2)
 [1] 17897
```

So 2019-01-01 is 17,897 days since the R time origin. We can find the time origin by subtracting this number from `d2`. Note that we can subtract numbers from dates, but not the other way round.

```
> d2 - 17897
 [1] "1970-01-01"
```

Datetimes

In data of type **datetime**, we store the same as in type date, plus hour, month and second. The preferred format is `yyyy-mm-dd hh:mm:ss`. Sometimes, an indication of the time zone is added at the end, as we will see below. Examples are CET (Central European Time), CEST (Central European Summer Time), GMT (Greenwich Mean Time) and UTC (Coordinated Universal Time). Datetime is also called **timestamp**.

POSIX (Portable Operating System Interface) is a "manufacturer-neutral" set of standard operating system interfaces based on the Unix operating system. `POSIXt` is an R class, based on POSIX. In practice, we can manage the `POSIXt` class in two ways:

- A `POSIXct` object is the number of seconds since the origin (`1970-01-01 00:00:00 CET`).
- A `POSIXlt` object is a list of time attributes (see below).

To show how this work, I enter a datetime as a string.

```
> dt1 = '2019-01-01 00:00:00'
> class(dt1)
 [1] "character"
```

Next, I turn it into a `POSIXct` object, finding the time origin as I did with dates.

```
> dt2 = as.POSIXct(dt1)
> class(dt2)
 [1] "POSIXct" "POSIXt"
> as.numeric(dt2)
 [1] 1546300800
> dt2 - 1546300800
 [1] "1970-01-01 01:00:00 GMT"
```

Now, I take a look at the `POSIXlt` version.

```
> dt3 = as.POSIXlt(dt1)
> dt3
 [1] "2019-01-01 GMT"
> unlist(dt3)
   sec min hour mday mon  year wday yday isdst  zone gmtoff
   "0" "0"  "0"  "1" "0" "119"  "2"  "0"   "0" "GMT"     NA
```

String data

Strings are managed in R by means of the class `character`. This includes not only sequences of **alphanumeric characters**, but also **white space** and **punctuation**. Beware that the **empty string** (`''`) is not the same as `NA`. It is a string of length zero.

`stringr` is a popular package for text manipulation, which I recommend. Almost all the `stringr` functions are available in plain R, but their names in `stringr` are intuitively clear, and the syntax, more specifically the order of the arguments, is consistent across functions. A clear introduction to

`stringr` can be found at `www.stringr.tidyverse.org`. For a more general presentation of string functions, I would recommend Chapter 14 of Wickham & Grolemund (2016).

The main functions (all vectorized) are:

- `str_length`: The length of a string.
- `str_sub`: Extract substrings from a character vector.
- `str_c`: Join multiple strings into a single string.
- `str_detect`: Detect the presence or absence of a pattern in a string.
- `str_extract_all`: Extract matching patterns from a string.
- `str_to_lower`: Convert to lowercase.
- `str_replace_all`: Replace matched patterns in a string.
- `str_split`: Split up a string into pieces.
- `str_trim`: Trim whitespace from start and end of string.

I show you some simple examples. I start with `str_length`.

```
> library(stringr)
> str_length(c('Donald Trump', 'Hillary Clinton'))
 [1] 12 15
> str_length('')
 [1] 0
```

With `str_sub`, we can extract **substrings** from a character vector. This is useful to manage dates, as shown below.

```
> dates = c('2016-10-06', '2015-08-19', '2016-01-30')
> year = str_sub(dates, 1, 4)
> year
 [1] "2016" "2015" "2016"
```

The function `str_sub` can also be used to replace substrings:

```
> str_sub(dates, 1, 4) = 'yyyy'
> dates
 [1] "yyyy-10-06" "yyyy-08-19" "yyyy-01-30"
```

With the function `str_c`, we **paste multiple strings** into a single string. The glue is specified with the parameter `sep`. The default is `sep=''`, but it is better to specify and name it to avoid confusion.

```
> str_c('Let us suppose', 'that this is not true', sep=' ')
 [1] "Let us suppose that this is not true"
```

Many methods of string data analysis are based on counting the occurrences of selected terms. Counting is preceded by **conversion to lowercase**, performed with the function `str_lowercase`.

```
> students = c('Donald', 'Pablo', 'Liudmila', 'Nana Yaa')
> str_to_lower(students)
 [1] "donald" "pablo" "liudmila" "nana yaa"
```

White space at the extremes of a string can create a problem when splitting a string into a bag of words (with `str_split`). White space is trimmed with the function `str_trim`.

```
> str_trim('Correlation is not causation ')
 [1] "Correlation is not causation"
```

With the function `str_detect`, we can detect the presence or absence of a **pattern** in a string. More specifically, this function returns a logical vector indicating, term by term, whether the pattern occurs in a character vector.

```
> str_detect(students, 'an')
 [1] FALSE FALSE FALSE TRUE
```

With the function `str_extract_all`, we can **extract matching patterns** from a string. When applying this function to a vector, it produces for each term a vector containing all the occurrences of the pattern. Typically, these vectors have different lengths, so they are packed in a list.

```
> str_extract_all(students, 'a')
 [[1]]
 [1] "a"

 [[2]]
 [1] "a"

 [[3]]
 [1] "a"

 [[4]]
 [1] "a" "a" "a" "a"
```

With the additional argument `simplify=TRUE`, this function pads the gaps with empty strings, so it can return a matrix:

```
> str_extract_all(students, 'a', simplify= TRUE)
     [,1] [,2] [,3] [,4]
[1,] "a"  ""   ""   ""
[2,] "a"  ""   ""   ""
[3,] "a"  ""   ""   ""
[4,] "a"  "a"  "a"  "a"
```

With the function `str_replace_all`, we can **replace matched patterns** in a string.

```
> str_replace_all(students, ' ', '-')
 [1] "Donald" "Pablo" "Liudmila" "Nana-Yaa"
```

While the third argument of `str_replace_all` (the replacement) has to be a single string, the second argument (the pattern) can be multiple. In the preceding example, we replaced single white space by a dash. Now, to replace either white space or the letter 'o', we set as the pattern to replace the regular expression 'o| '. Note that, in R (as in many languages), the vertical bar means 'OR'.

```
> str_replace_all(students, 'o| ', '-')
 [1] "D-nald" "Pabl-" "Liudmila" "Nana-Yaa"
```

The function `str_split` **splits up a string into pieces**. This is one way to transform a string into a **bag of words**, that is, a vector whose terms are the words contained in the string. Note that this function also returns a list, because the bags of words extracted from a character vector have different lengths.

```
> str_split('Correlation is not causation', ' ')
 [[1]]
 [1] "Correlation" "is" "not" "causation"
```

Regular expressions

Some of the transformations performed in the preprocessing stage are dramatically simplified by using **regular expressions**. A regular expression is a pattern which describes a set of strings. We have seen an example of this in the preceding section, when using the vertical bar within a pattern.

Among the regular expressions, **character classes** are easy to understand. For instance, `[0-9]` stands for any digit, and `[A-Z]` for any capital letter. The square brackets indicate *any* of the characters enclosed.

I show how this works with some simple examples.

```
> bio = 'I was born in 1954'
> str_replace_all(bio, '[a-z]', 'x')
 [1] "I xxx xxxx xx 1954"
> str_replace_all(bio, '[a-zA-Z]', 'x')
 [1] "x xxx xxxx xx 1954"
> str_replace_all(bio, '[0-9]', 'x')
 [1] "I was born in xxxx"
```

Character classes get more powerful when complemented with **quantifiers**. For instance, followed by a plus sign (+), a character class indicates a sequence of any length. So, `[0-9]+` indicates any sequence of digits, therefore any number. We can also specify the minimum and maximum length of the sequence, as in the second example below.

```
> str_replace_all(bio, '[a-zA-Z]+', 'x')
 [1] "x x x x 1954"
> str_replace_all(bio, '[0-9]{1,4}', 'x')
 [1] "I was born in x"
```

A simple clean way of getting a bag of words is as follows.

```
> str_extract_all(bio, '[a-zA-Z0-9]+')
 [[1]]
 [1] "I" "was" "born" "in" "1954"
```

Regular expressions are a whole chapter of programming, with entire books, such as Friedl (2007), devoted to them. Beginners may also try the `regexr` website at `www.regexr.com`, which makes fun of learning regular expressions. To learn how to use regular expressions within `stringr` functions, you can look at the tutorial in `www.stringr.tidyverse.org/articles/regular-expressions.html`.

Special characters

Text imported from PDF or HTML documents, or from devices like mobile phones, may contain **special characters** like the n-dash (–), the left/right quotation marks (“, ‘, etc), or the three-dot character (…), which is better to control, to avoid confusion. To keep this note short, I do not develop this point here, but mind that, if you capture text data on your own, you will probably find some of this in your data. Even if the documents are expected to be in English, they can be contaminated by other languages: Han characters, German umlaut, Spanish eñe, etc.

Another source of trouble is that these special symbols can be encoded by different computers or different text editors in different ways. The typical **encodings** in the Western world are UTF-8 and Latin-1. R can deal with these two, but may have trouble with other encodings. I do not discuss encodings in this course. If you are interested, you may take a look at Korpela (2006).

References

1. JEF Friedl (2007), *Mastering Regular Expressions*, O’Reilly.
2. JK Korpela (2006), *Unicode Explained*, O’Reilly.
3. H Wickham & G Grolemund (2016), *R for Data Science*, O’Reilly. Free access at `r4ds.had.co.nz`.