Formatting Tasks with Fm()

by Andri Signorell

Helsana Versicherungen AG, Health Sciences, Zurich
HWZ University of Applied Sciences in Business Administration, Zurich

andri@signorell.net

January, 1th, 2026

Formatting numbers can be a nightmare in R, and using base resources such tasks can be extremely time-consuming. The reason for this is the multitude of available options (e.g. `format()`, `formatC()`, `sprint()`, `symbol()`, `prettyNum()`, `strftime()` etc.), which often makes it difficult to find the right one. The functionality often overlaps considerably, lacking consistency. Some functions support certain representations, others do not. Other desirable format options again cannot be found at all.

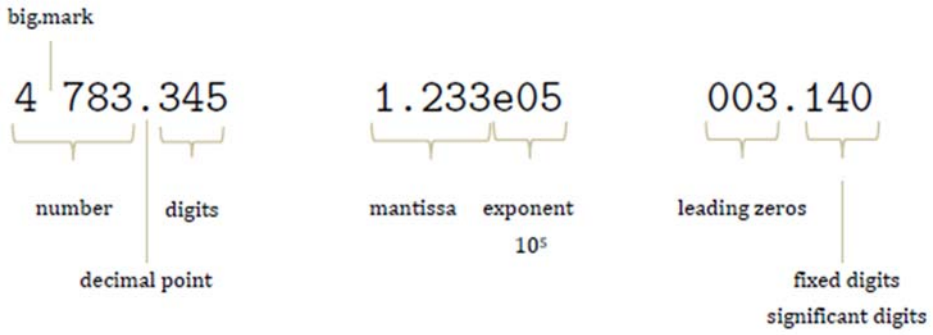
`Fm()` is a general-purpose formatting function for R objects, designed for statistical reporting and publication output. It provides a unified interface for formatting numeric values, dates, text, and tabular objects, with support for reusable format styles.

Note: For all the examples in this document, `library(DescTools)` must be declared.

1 Formatting Numbers

1.1 Basic Formatting

Let us take a quick look at the aspects of representing numbers. Firstly, there is the question of precision. Note that R by default uses "significant" number of digits to display numeric values, which can be set by `options(digits=4)`. Note, that a user may often want to have a FIXED number of digits (consider e.g. p-values, where the decimal places in 1.234e-17 are hardly of any interest).
Next we have to decide, when we want to switch to exponential representation. This setting too can be set by an option: `options(scipen=7)` would cause the system to switch to scientific notation for numbers ≥ 10^7.



We should be aware that formatting means converting a numeric value into a string representation. Calculating with strings can be very troublesome (think of removing big marks). This entails that formatting should only be done after all calculations or modelling processes have been completed.

So how can we use `Fm()` to get our numbers in shape? Although the function has a large number of arguments, in most cases only a few specific settings are required.

Usage

```
Fm(x, digits = NULL, sci = NULL, big.mark = NULL,
       ldigits = NULL, zero.form = NULL, na.form = NULL,
       fmt = NULL, align = NULL, width = NULL, lang = NULL,
       eps = NULL, outdec = NULL, ...)
```

The first example addresses the number of digits, the transition to scientific representation and the use of big marks. It uses a space as big mark, align the numbers on the position of the "e", flip to scientific notation for numbers < 10^{-2} and > 10^4 and uses 3 fixed digits for all numbers.

`x`	an atomic numerical, typically a vector of real numbers or a matrix of numerical values. Factors will be converted to strings.	

`digits`	*integer*, the desired (fixed) number of digits after the decimal point. Unlike `formatC()` you will always get this number of digits even if the last digit is 0. The result is rounded using `round()`.
	Negative numbers of digits round to the specific power of ten (`digits=2` would round to the nearest hundred).
		`Fm(3.141593, digits=3)`
		`## 3.142`
		`Fm(0.031415, digits=3)`
		`## 0.031`
		`Fm(3142.2, digits=-2)`
		`## 3100`
		`Fm(3.10012, digits=3)`
		`## 3.100`
		`Fm(c(3.1422, -1.5), digits=0)`
		`## 3 -2`

`ldigits`	*nonnegative integer*, number of leading zeros. `ldigits=3` would make sure that at least 3 digits on the left side will be printed.
	Setting `ldigits=0` will remove a leading zero for values in the interval of (1, -1) and yield a result like .452 for 0.452.
	The default `NULL` will leave the numbers as they are (meaning at least one 0 digit). Negative values are ignored.
		`Fm(3.1415, ldigits=3, digits=2)`
		`## 003.14`
		`Fm(3.1415, ldigits=0, digits=2)`
		`## 3.14`
		`Fm(0.3141, ldigits=0, digits=2)`
		`## .31`

`sci`	*integer,* the power of 10 to be set when deciding to print numeric values in exponential notation. Fixed notation will be preferred unless the number is larger than 10^{scipen}. If just one value is set for `scipen` it will be used for the left border $10^{-scipen}$ as well as for the right one (10^{scipen}). A negative and a positive value can also be set independently. Default is `getOption("scipen")`, whereas `scipen=0` is overridden.	

`big.mark`	*character,* if not empty used as mark between every 3 decimals before the decimal point. Default is `""` (none).	`Fm(3141, big.mark=" ")`
		`## 3 141.000`
		`Fm(3141.12, big.mark=",", digits=0)`
		`## 3,141`

`outdec`	*character*, specifying the decimal mark to be used. If not provided, the default is given by `getOption("OutDec")`.	`Fm(3141.593,`
		`  big.mark="'", outdec = ",")`
		`## 3'141,593`

Using options for big.mark, sci, digits, and outdec.

1.2 Specific Formats

Sometimes more variability is needed to display numeric values. For such cases of more specific formatting of numerical values, there is the `fmt` argument. It is very flexible and is used to generate a variety of different formats, such as percentages, engineering representation or p-values. If x is a date it serves for the specific date/time format codes (see 2.1).

`fmt`	*character*, interpreted as format string, allowing to flexibly define special formats or object of `style` class, consisting of a list of arguments accepted by `Fm()`.
`p_eps`	*number*, the tolerance used for formatting p values, those less than `p_eps` are formatted as `"< [p_eps]"`. Default is `0.001`.

For the most frequently used formats there are the following special codes available:

`%`	**percent** will divide the given number by 100 and append the %-sign (without a separator). Digits will be set to 1, if not provided otherwise.	
`e`	**scientific** forces scientific representation of x, e.g. 3.141e-05. The number of digits, alignment and zero values are further respected. Digits defaults to 3.	

`p`	**p-value** returns a numeric value in p-value format. `p_eps` defines the threshold to e.g. switch to a < 0.001 representation (more detailed comments below).	`Fm(0.003, fmt="p")`
		`## .0030`
		`Fm(0.00034, fmt="p", eps=0.001)`
		`## < 0.001`
		`Fm(1.2, fmt="p")`
		`## <NA>`
		`Fm(-0.2, fmt="p")`

```
## <NA>
```

*	**significance** will produce a significance representation of a p-value consisting of * and . , while the breaks are set according to the used defaults e.g. in `lm()` as
	[0, 0.001] = ***
	(0.001, 0.01] = **
	(0.01, 0.05] = *
	(0.05, 0.1] = .
	(0.1,1] =
		`Fm(0.082, fmt="*")`
		`## .`
		`Fm(0.003, fmt="*")`
		`## **`
		`Fm(1.2, fmt="*")`
		`## <NA>`
		`Fm(-0.2, fmt="*")`
		`## <NA>`

p*	**p-value AND stars** will produce p-value and significance stars	
eng	**engineering** forces scientific representation, restricting to powers that are a multiple of 3.	
engabb	**engineering abbreviation** same as eng, but replaces the exponential representation by codes, e.g. M for mega (1e6). See d.prefix.	
frac	**fractions** will (try to) convert numbers to fractions. So 0.1 will be displayed as 1/10. See `fractions()`.	

When `fmt = "p"`, numeric input is interpreted as p-values and formatted for reporting in scientific tables and manuscripts. Only values in the interval from 0 to 1 are considered valid; values outside this range are returned as missing.
Very small p-values are reported using a less-than notation based on a configurable reporting threshold (p_eps), which defaults to 0.001. Values equal to or below this threshold are not printed numerically. Exact zeros are never shown. P-values greater than the threshold are reported numerically using a fixed number of decimal places, controlled by the `digits` argument. A p-value equal to one is reported as "1" without trailing decimals.
By default, p-values are printed with a leading zero before the decimal point. APA-style output without a leading zero can be obtained by setting `ldigits = 0`, which affects only the visual representation. Comparison operators are formatted with surrounding spaces for typographic clarity. Missing values are propagated unchanged.
The formatter is designed for publication-ready output and intentionally avoids exposing machine-level numerical precision, such as machine epsilon values, which are rarely meaningful in scientific reporting.

1.3 Special values

Missing values sometimes require special representation. This is made possible by the argument `na.form`. All missing values are given the specified form as soon as the argument is assigned. The same applies to 0 values with the argument `zero.form`.

`na.form`	*character*, string specifying how NAs should be specially formatted. If set to NULL (default) no special action will be taken.	`Fm(c(1, NA, 8), na.form=".", digits=0)` `## 1 . 8`
`zero.form`	*character*, string specifying how zeros should be specially formatted. Useful for pretty printing 'sparse' objects. If set to NULL (default) no special action will be taken.	`Fm(c(1, 0, 8), zero.form="-", digits=0)` `## 1 - 8`

1.4 Alignment

Alignment can be challenging. A simple and robust approach in console-based environments that use fixed-width (monospace) fonts is to rely on spaces for alignment, ensuring predictable and reproducible output across platforms and devices.

The following arguments provide fine-grained control over how character strings are aligned and displayed within a fixed width, making it straightforward to produce neatly formatted console output.

`align`	the character on whose position the strings will be aligned. Left alignment can be requested by setting `align = "\\l"`, right alignment by `"\\r"` and center alignment by `"\\c"`. Mind the backslashes, as if they are omitted, strings would be aligned to the **character** l, r or c respectively. The default is `NULL` which would just leave the strings as they are. This argument is sent directly to `StrAlign()` as argument `sep`.
`width`	integer, the defined fixed width of the strings.

1.5 Combinations of multiple numbers

FmCI

`FmCI()` is a lightweight helper function in *DescTools* for formatting confidence intervals in a concise and flexible way. It builds on `Fm()`, which provides consistent numeric rounding and formatting, and adds a thin presentation layer that turns numeric results into publication-ready text.
The core idea of `FmCI()` is to centralize confidence-interval formatting. Instead of repeatedly combining `Fm()`, `paste()`, or `sprintf()`, `FmCI()` produces consistent output with minimal code while remaining fully customizable through a user-defined template.
By default, `FmCI()` automatically interprets the structure of its input. If x has length three, it is assumed to contain a point estimate followed by lower and upper confidence limits, and the result is formatted as estimate [lower, upper].
If x has length two, it is interpreted as confidence limits only and formatted as [lower, upper].
This behavior covers the most common reporting situations without requiring any additional arguments.

The template argument allows full control over the output layout and overrides the default behavior. Any format supported by `gettextf()` can be used, making `FmCI()` adaptable to different journal styles and reporting conventions.

Arguments

- x: A numeric vector of length 2 or 3. Length 3 is interpreted as point estimate, lower, and upper confidence limits; length 2 as lower and upper confidence limits only.
- template: An optional character string passed to `gettextf()` that defines the output format. If `NULL`, a default template is chosen based on the length of x.
- ...: Additional arguments forwarded to `Fm()`, allowing control over rounding, digits, and numeric formatting.

Despite its simplicity, `FmCI()` is intentionally designed as a small but expressive formatting primitive that reduces boilerplate code while ensuring consistent and reproducible presentation of confidence intervals.

```
FmCI(c(0.42, 0.30, 0.55))                    # estimate and confidence interval

## 0.42 [0.30, 0.55]

FmCI(c(0.30, 0.55))                          # confidence interval only

## [0.30, 0.55]

FmCI(c(0.42, 0.30, 0.55),                    # custom output template
     template = "%s (%s-%s)", digits = 1)

## 0.4 (0.3-0.6)
```

1.6 Vectorizing and Objects

The function is vectorized, so it takes a vector and applies any formats to all the elements in the vector (using the usual recycling rule – is it ????).

x	an atomic numerical, typically a vector of real numbers or a matrix of numerical values. Factors will be converted to strings.

1.7 Using Styles

name	a name for a defined style
label	a description for a defined style

fmt can as well be an object of class fmt consisting of a list out of the arguments above. This allows to store and manage the full format in variables or as options (in DescToolsOptions()) and use it as format template subsequently.

Finally fmt can also be a function in x, which makes formatting very flexible.

New formats can be created by means of as.fmt(). This works quite straight on. We can use any of the arguments from Fm() and combine them to a list.
The following code will define a new format template named "myNumFmt" of the class "fmt". Provided to Fm() this will result in a number displayed with 2 fixed digits and a comma as big mark:

myNumFmt <- as.fmt(digits=2, big.mark=",")
Fm(12222.89345, fmt=myNumFmt) = 12,222.89
The latter returns the same result as if the arguments would have been supplied directly:
Fm(12222.89345, digits=2, big.mark=",").

Many report functions (e.g. TOne()) in DescTools use three default formats for counts (named "abs"), numeric values ("num") and percentages ("per"). These formats can be set by the user as options (see DescToolsOptions(). For other purposes any number of any named formats can be defined.

Style() is used to access and edit already defined format definitions. It can directly adapt defined properties and returns the format template. Style("num", digits=1, sci=10) will use the current version of the numeric format and change the digits to 1 and the threshold to switch to scientifc presentation to numbers >1e10 and <1e-10.

Styles() returns all found style definitions.

The formats can as well be organized as options. DescToolsOptions("fmt") would display the currently defined formats. This mechanic works analogously to the options() procedure of base R. So to store the current settings we can use

opt <- DescToolsOptions("fmt")
... do some stuff like redefining the global formats ...
DescToolOptions(opt)
The last command resets the options and so we have again the initial definitions for the format templates.

2 Formatting Dates

2.1 Display Calendar/Time Data

The argument `fmt` can be used to format `Date` and `POSIXct` objects using a custom, ISO-8601–inspired token syntax similar to .NET or Moment.js.
These format codes (e.g. d, M and y for day, month or year) are more intuitive than the C format codes. Repeating the specific code defines the degree of abbreviation. So the format 'yyyy-mm-dd' would yield a date as 2020-10-12. Weekdays and month names can be displayed in the current locale or in English.

`d`	day of the month without leading zero (1 - 31)
`dd`	day of the month with leading zero (01 - 31)
`ddd`	abbreviated name for the day of the week (e.g. Mon) in the current user's language
`dddd`	full name for the day of the week (e.g. Monday) in the current user's language
`M`	month without leading zero (1 - 12)
`MM`	month with leading zero (01 - 12)
`MMM`	abbreviated month name (e.g. Jan) in the current user's language
`MMMM`	full month name (e.g. January) in the current user's language
`y`	year without century, without leading zero (0 - 99)
`yy`	year without century, with leading zero (00 - 99)
`yyyy`	year with century
`H/HH`	Hour in 24h format, one digit / two digits
`h/hh`	Hour in 12h format, one digit / two digits, note that in this case t must be set also to ensure uniqueness.
`t/tt`	Adds AM/PM description (one/two characters)
`m/mm`	Minutes one digit / two digits
`s/ss`	Seconds one digit / two digits
`lang`	*character*, optional value setting the language for the months and daynames. Can be either `"local"` for current locale or `"en"` for English. If left to `NULL`, the DescTools option `"lang"` will be searched for and if not found `"local"` will be taken as default.

```
x <- as.Date(c(
       "2026-01-02","2018-10-17"))

Fm(x, fmt="...")
```

```
d.MM.yyyy
##  2.01.2026     17.10.2018
```

```
dd.MM.yyyy
##  02.01.2026     17.10.2018
```

```
ddd, dd.MM.yyyy
## Fr, 02.01.2026   Mi, 17.10.2018
```

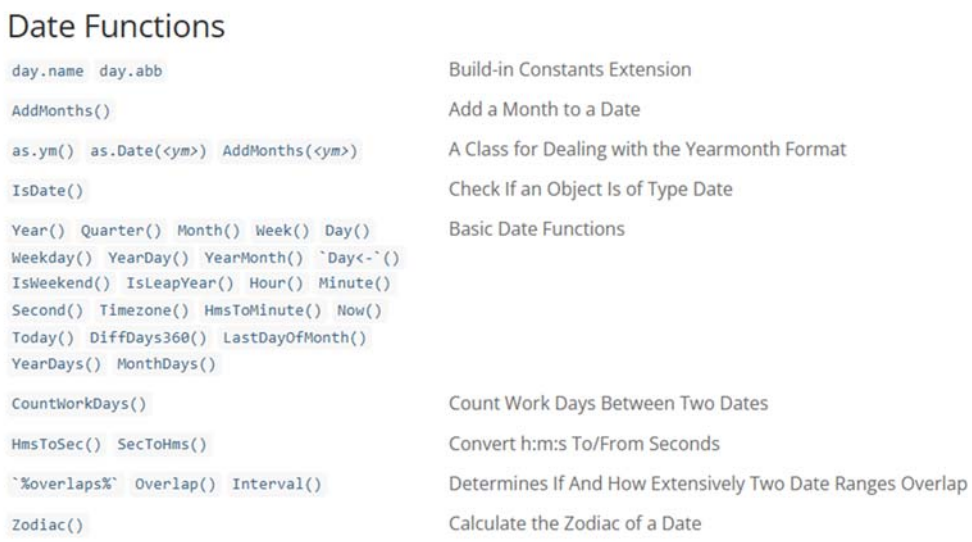
```
y <- as.POSIXct(
        "2026-01-02 21:14:12 CET")
```

```
H-m-ss tt
## 21-14-12 PM
```

2.2 Date functions

In the broadest sense, special properties of calendar data, such as the year, month or day, are also relevant to the display. However, this information often forms the basis for further calculations, which are then sensibly implemented as functions. Why bother with the

complexity of a sophisticated time object when all you need is the difference between two calendar years? Properties such as day of the year and similar have definitely little to do with representation.
In DescTools, such tasks are therefore solved with (extraction) functions. The following are available for this purpose:



3 Formatting Higher Dimensional Objects

3.1 Matrices

3.2 data.frames

The built-in data set *HairEyeColor* has the class *table*. Let's turn this table into a case-by-case data frame as a base for the subsequent analysis. `Untable` does this job.

Untable

```
d.col <- Untable(HairEyeColor)
head(d.col, 3)

## Hair   Eye  Sex
## 1 Black Brown Male
## 2 Black Brown Male
## 3 Black Brown Male
```

From here we can start tabulating again. The simplest case is to tabulate a single vector. The function `table` yields the absolute frequencies and `prop.table` the proportions:

table
prop.table

```
table(d.col$Hair)

## Black Brown   Red Blond
##   108   286    71   127
```

```
prop.table(table(d.col$Hair))

##     Black     Brown       Red     Blond
## 0.1824324 0.4831081 0.1199324 0.2145270
```

4 References

(1) Agresti A. (2002) Categorical Data Analysis. John Wiley & Sons.

(2) Dalgaard P. (2008) Introductory Statistics with R (2. Aufl.), London, UK: Springer.

(3) Friendly M. (2013) Working with categorical data with R and the vcd and vcdExtra packages, York University, Toronto.
http://cran.r-project.org/web/packages/vcdExtra/vignettes/vcd-tutorial.pdf

(4) One-Way Frequency Tables using SAS PROC FREQ © TexaSoft, 2006
http://www.stattutorials.com/SAS/TUTORIAL-PROC-FREQ-1.htm

(5) Presnell B. (2011) Course Notes sta4504-2011sp,
http://www.stat.ufl.edu/~presnell/Courses/sta4504-2011sp/Notes/icda-notes-3x2.pdf

(6) SAS/STAT® 9.2 User's Guide, Second Edition, The FREQ Procedure (Book Excerpt) (2009)
http://support.sas.com/documentation/cdl/en/statugfreq/63124/PDF/default/statugfreq.pdf