Tables in R Markdown

Duncan Murdoch

13/05/2019

Contents

1 Introduction **2**

2 Reference **3**
2.1 Function syntax . . . 4
2.1.1 `tabular()` . . . 4
2.1.2 `format()`, `print()`, `toLatex()` . . . 4
2.1.3 `as.matrix()`, `write.csv.tabular()`, `write.table.tabular()` . . . 5
2.1.4 `as.tabular()` . . . 5
2.1.5 `table_options()`, `booktabs()` . . . 5
2.1.6 `latexNumeric()` . . . 8
2.2 Operators . . . 8
2.2.1 $e_1 + e_2$ . . . 8
2.2.2 $e_1 * e_2$ . . . 9
2.2.3 $e_1 \sim e_2$ . . . 9
2.2.4 $e_1 = e_2$ . . . 9
2.3 Terms in Formulas . . . 9
2.3.1 Closures or other functions . . . 10
2.3.2 Factors . . . 10
2.3.3 Logical vectors . . . 10
2.3.4 Language Expressions . . . 11
2.3.5 Other vectors . . . 11
2.4 "Pseudo-functions" . . . 11
2.4.1 `Format()` . . . 11
2.4.2 `.Format()` . . . 12
2.4.3 `Heading()` . . . 12
2.4.4 `Justify()` . . . 13
2.4.5 `Percent()` . . . 14
2.4.6 `Arguments()` . . . 15
2.4.7 `DropEmpty()` . . . 15
2.5 Formula Functions . . . 16
2.5.1 `All()` . . . 16
2.5.2 `AllObs()`, `RowNum()` . . . 17
2.5.3 `Hline()` . . . 18
2.5.4 `Literal()` . . . 19
2.5.5 `PlusMinus()` . . . 19
2.5.6 `Paste()` . . . 19
2.5.7 `Factor()`, `RowFactor()` and `Multicolumn()` . . . 20

3 Further Details **24**
3.1 Formatting . . . 24

3.2 Missing Values . . . 25
3.3 Subsetting and Joining Tables . . . 26
3.4 knitr, rmarkdown and kableExtra support . . . 28
3.5 Captions, labels, etc. . . . 28

4 Acknowledgments **29**

References **29**

1 Introduction

This vignette was built using `tables` version 0.9.3. It is intended to show the same content as the `tables.pdf` vignette that was written in Sweave, but with R Markdown source code. This has allowed a few simplifications; see Section 3.4 for a description of them.

It is a short introduction to the `tables` package. Inspired by my 20 year old memories of SAS PROC TABULATE, I decided to write a simple utility to create nice looking tables in Sweave documents. (It now also works in R Markdown documents, as this vignette illustrates.) For example, we might display summaries of some of Fisher's iris data using the code

```
tabular( (Species + 1) ~ (n=1) + Format(digits=2)*
         (Sepal.Length + Sepal.Width)*(mean + sd), data=iris )
```

```
##
##                Sepal.Length      Sepal.Width
##  Species    n   mean         sd   mean        sd
##  setosa      50 5.01         0.35 3.43        0.38
##  versicolor  50 5.94         0.52 2.77        0.31
##  virginica   50 6.59         0.64 2.97        0.32
##  All        150 5.84         0.83 3.06        0.44
```

You can also pass the output through the `toLatex()` function to produce LaTeX output, which when processed by `pdflatex` will produce the following table:

		Sepal.Length		Sepal.Width	
Species	n	mean	sd	mean	sd
setosa	50	5.01	0.35	3.43	0.38
versicolor	50	5.94	0.52	2.77	0.31
virginica	50	6.59	0.64	2.97	0.32
All	150	5.84	0.83	3.06	0.44

However, if you are using `rmarkdown` or `knitr` (as this document does), `toLatex()` is not necessary. Just execute

```
table_options(knit_print = TRUE)
```

at the start of your document, and conversion to LaTeX will be done automatically when needed.

If you prefer the style of table that the LaTeX `booktabs` package (Fear 2005) produces, you can choose that style instead. I mostly like it, so I have used

```
booktabs()
```

for the rest of this document. This gives

		Sepal.Length		Sepal.Width	
Species	n	mean	sd	mean	sd
setosa	50	5.01	0.35	3.43	0.38
versicolor	50	5.94	0.52	2.77	0.31
virginica	50	6.59	0.64	2.97	0.32
All	150	5.84	0.83	3.06	0.44

Details on `booktabs()` are given in section 2.1.5 below.

There is also the `toHTML` function and `html.tabular` method for the `Hmisc::html()` generic; they produce output in HTML format. Finally, see section 2.1.3 for other output formats.

The idea of a table in the `tables` package is a rectangular array of values, with each row and column labelled, and possibly with groups of rows and groups of columns also labelled. These arrays are specified by "table formulas".

Table formulas are R formula objects, with the rows of the table described before the tilde (`"~"`), and the columns after. Each of those is an expression containing `"*"`, `"+"`, `"="`, as well as functions, function calls and variables, and parentheses for grouping. There are also various directives included in the formula, entered as "pseudo-functions", i.e. expressions that look like function calls but which are interpreted by the `tabular()` function.

For example, in the formula

```
(Species + 1) ~ (n=1) + Format(digits=2)*
         (Sepal.Length + Sepal.Width)*(mean + sd)
```

the rows are given by `(Species + 1)`. The summation here is interpreted as concatenation, i.e. this says rows for `Species` should be followed by rows for `1`.

In the `iris` dataframe, `Species` is a factor, so the rows for it correspond to its levels.

The `1` is a place-holder, which in this context will mean "all groups".

The columns in the table are defined by

```
(n=1) + Format(digits=2)*(Sepal.Length + Sepal.Width)*(mean + sd)
```

Again, summation corresponds to concatenation, so the first column corresponds to `(n=1)`. This is another use of the placeholder, but this time it is labelled as `n`. Since we haven't specified any other statistic to use, the first column contains the counts of values in the dataframe in each category.

The second term in the column formula is a product of three factors. The first, `Format(digits=2)`, is a pseudo-function to set the format for all of the entries to come. (For more on formats, see section 2.4.1 below.) The second factor, `(Sepal.Length + Sepal.Width)`, is a concatenation of two variables. Both of these variables are numeric vectors in `iris`, and they each become the variable to be analyzed, in turn. The last factor, `(mean + sd)` names two R functions. These are assumed to be functions that operate on a vector and produce a single value, as `mean` and `sd` do. The values in the table will be the results of applying those functions to the two different variables and the subsets of the dataset.

2 Reference

For the examples below we use the following definitions:

```
set.seed(100)
X <- rnorm(10)
X
```

```
##  [1] -0.50219235  0.13153117 -0.07891709  0.88678481
```

```
##  [5]  0.11697127  0.31863009 -0.58179068  0.71453271
##  [9] -0.82525943 -0.35986213
```

```
A <- sample(letters[1:2], 10, rep=TRUE)
A
```

```
##  [1] "a" "b" "b" "a" "b" "a" "a" "b" "b" "a"
```

```
F <- factor(A)
F
```

```
##  [1] a b b a b a a b b a
## Levels: a b
```

2.1 Function syntax

2.1.1 `tabular()`

```
tabular(table, ...)
tabular.default(table, ...)
tabular.formula(table, data=parent.frame(), n, suppressLabels=0, ...)
```

The `tabular` function is a generic function. The default method uses `as.formula()` to try to convert the `table` argument to a formula, then passes it and all the other arguments to `tabular.formula()` method, which does most of the work. That method has 4 arguments plus `...`, but usually only the first two are used, and a warning is issued if anything is passed in the `...` arguments.

- `table`: The `table` argument is the table formula, described in detail below.
- `data`: The `data` argument is a dataframe or environment in which to look for the data referenced by the table.
- `n`: The `tabular` function needs to know the length of vectors on which it operates, because some formulas (e.g. `1 ~ 1`) contain no data. Normally `n` is taken as the number of rows in `data`, or the length of the first referenced object in the formula, but sometimes the user will need to specify it. Once specified, it can't be modified: all data in the table should be the same length.
- `suppressLabels`: By default, `tabular` adds a row or column label for each term, but this does sometimes make the table messy. Setting `suppressLabels` to a positive integer will cause that many labels to be suppressed at the start of each term. The pseudo-function `Heading()` can achieve the same effect, one term at a time.

The value returned is a list-mode matrix corresponding to the entries in the table, with a number of attributes to help with formatting. See the `?tabular` help page for more details.

2.1.2 `format(), print(), toLatex()`

```
format(x, digits=4, justification="n", ...)
print(x, ...)
toLatex(x, file="", options=NULL,  ...)
```

The `tables` package provides methods for the `format()`, `print()` and `utils::toLatex()` generics. The arguments are:

- `x`: The tabular object returned from `tabular()`.
- `digits`: The default number of digits to use when formatting.
- `justification`: The default text justification to use when printing. For text display, the recognized values are `"n"`, `"l"`, `"c"`, `"r"`, standing for none, left, center and right justification respectively. For LaTeX the justification is specified via the `table_options()` function (section 2.1.5).

- `file`: The default method for the `Hmisc::latex()` generic writes the LaTeX code to a file; `latex.tabular()` can optionally do the same, but it defaults to writing to screen, for use in Sweave documents like this one.
- `options`: A list of options to pass to `table_options()`. These will be set only for the duration of the call to `toLatex()`.

2.1.3 as.matrix(), write.csv.tabular(), write.table.tabular()

```
as.matrix(x, format = TRUE,
    rowLabels = TRUE, colLabels = TRUE, justification = "n", ...)
write.csv.tabular(x, file = "", justification = "n", row.names=FALSE,
    write.options=list(), ...)
write.table.tabular(x, file="",
    justification = "n", row.names=FALSE, col.names=FALSE,
    write.options=list(), ...)
```

These functions export tables for further computations. The arguments are:

- `x`: The tabular object.
- `format`: Whether to format the entries. See the help page for alternatives.
- `rowLabels, colLabels`: If formatting, whether to include the labels or not.
- `justification`: The default text justification to use when formatting.
- `file`: Where to write the output.
- `row.names,col.names, write.options`: Additional parameters to pass to `write.csv()` or `write.table()`.

2.1.4 as.tabular()

```
as.tabular(x, ...)
as.tabular.default(x, like=NULL, ...)
as.tabular.data.frame(x, ...)
```

These functions create tables from existing matrices or dataframes of values. The dimnames of the input are used to construct default row and column names. If more elaborate labelling is wanted, use a `tabular` object as the `like` argument. The labelling for `like` will be used on the newly constructed result.

2.1.5 table_options(), booktabs()

The `table_options()` function sets a number of formatting defaults for the `toLatex()` method:

- `justification`: This is the default justification for data columns and their headers. Any justification string will be accepted; it should be one that the LaTeX `\tabular` environment (or substitute) accepts. If a vector of strings is specified they will be recycled across the columns of the table.
- `rowlabeljustification`: This is the default justification for row labels. A vector of strings will be recycled across the row label columns.
- `tabular`: The environment to use in LaTeX. Alternatives to `"tabular"` such as `"longtable"` can be used here. Those often also need modifications within the table; the `Literal()` (section \ref{sec:Literal‘) function may be helpful.
- `toprule, midrule, bottomrule`: The LaTeX macros to draw the top, middle and bottom lines in the table. By default these are all `"\\hline"`.
- `titlerule`: An optional LaTeX macro to draw a line under multicolumn titles.
- `doBegin, doHeader, doBody, doFooter, doEnd`: These logical values control the inclusion of specific parts of the output table.

The defaults are

```
## $justification
## [1] "c"
##
## $rowlabeljustification
## [1] "l"
##
## $tabular
## [1] "tabular"
##
## $toprule
## [1] "\\hline"
##
## $midrule
## [1] "\\hline"
##
## $bottomrule
## [1] "\\hline"
##
## $titlerule
## NULL
##
## $doBegin
## [1] TRUE
##
## $doHeader
## [1] TRUE
##
## $doBody
## [1] TRUE
##
## $doFooter
## [1] TRUE
##
## $doEnd
## [1] TRUE
##
## $knit_print
## [1] TRUE
##
## $latexleftpad
## [1] TRUE
##
## $latexrightpad
## [1] TRUE
##
## $latexminus
## [1] TRUE
##
## $doHTMLheader
## [1] FALSE
##
## $doCSS
## [1] FALSE
##
```

```
## $doHTMLbody
## [1] FALSE
##
## $CSS
## [1] "<style>\n#ID .center { \n  text-align:center;\n  background-color: aliceblue;\n}\n</style>"
##
## $HTMLhead
## [1] "<!DOCTYPE html>\n<html>\n<head>\n<meta charset=\"CHARSET\">\n"
##
## $HTMLbody
## [1] "<body>\n"
##
## $HTMLattributes
## [1] "class=\"Rtable\""
##
## $HTMLcaption
## NULL
##
## $HTMLfooter
## NULL
##
## $HTMLleftpad
## [1] TRUE
##
## $HTMLrightpad
## [1] TRUE
##
## $HTMLminus
## [1] TRUE
```

Some options only apply to HTML output; see the help page `?table_options` for details.

If you are using the LaTeX `booktabs` package, the `booktabs()` function will set different options. Currently those are:

```
## $toprule
## [1] "\\toprule"
##
## $midrule
## [1] "\\midrule"
##
## $bottomrule
## [1] "\\bottomrule"
##
## $titlerule
## [1] "\\cmidrule(lr)"
```

The earlier table of iris data was produced using

```
saved.options <- table_options()
invisible(booktabs())
tabular( (Species + 1) ~ (n=1) + Format(digits=2)*
         (Sepal.Length + Sepal.Width)*(mean + sd), data=iris )
```

We can use the `doXXXX` options to insert raw LaTeX into a table:

```
toLatex(tabular(Species ~ (n=1) + Format(digits=2)*
         (Sepal.Length + Sepal.Width)*(mean + sd), data=iris),
      options=list(doFooter=FALSE, doEnd=FALSE))
cat("\\ \\\\ \\multicolumn{6}{l}{
\\textit{Overall, we see the following: }} \\\\
\\ \\\\")
toLatex(tabular(1 ~ (n=1) + Format(digits=2)*
         (Sepal.Length + Sepal.Width)*(mean + sd), data=iris),
      options=list(doBegin=FALSE, doHeader=FALSE))
```

		Sepal.Length		Sepal.Width	
Species	n	mean	sd	mean	sd
setosa	50	5.01	0.35	3.43	0.38
versicolor	50	5.94	0.52	2.77	0.31
virginica	50	6.59	0.64	2.97	0.32
Overall, we see the following:					
All	150	5.84	0.83	3.06	0.44

Note that we need explicit `toLatex()` calls to access these options; in turn, that means the `knitr` chunk options require `results = "asis"`.

2.1.6 `latexNumeric()`

```
latexNumeric(chars, minus = TRUE, leftpad = TRUE, rightpad=TRUE,
                        mathmode = TRUE)
```

The `latexNumeric()` function converts character representations of numbers into a format suitable for display in LaTeX documents. There are two goals:

- If `chars` is a vector with constant width, then the output will also be constant width. This means the default centering used in `tabular()` will not misalign decimal points (if they were aligned in `chars`).
- Minus signs will be displayed with the proper symbol rather than a hyphen.

The arguments are:

- `chars`: A character vector of formatted numeric values.
- `minus`: Whether to pad positive cases with spacing of the same width as a minus sign. If `TRUE` and some entries are negative, then all positive entries will be padded.
- `leftpad, rightpad`: Whether to pad cases that have leading or trailing blanks with spacing matching a digit width per space. If `leftpad=TRUE`, leading blanks will be converted to spaces the same width as a digit 0. (If `minus=TRUE`, one leading blank may have been consumed in the sign padding.) The `rightpad` argument handles trailing blanks similarly.
- `mathmode`: Whether to wrap the result in dollar signs, so LaTeX will render minus signs properly.

2.2 Operators

2.2.1 $e_1 + e_2$

Summing two expressions indicates that they should be displayed in sequence. For rows, this means e_1 will be displayed just above e_2; for columns, e_1 will be just to the left of e_2.

Example:

```
tabular(F + 1 ~ 1)
```

F	All
a	5
b	5
All	10

2.2.2 $e_1 * e_2$

Multiplying two expressions means that each element of e_1 will be applied to each element of e_2. If e_1 is a factor, then e_2 will be displayed for each element of it. NB: $*$ has higher precedence than $+$ and evaluation proceeds from left to right. The expression $(e_1 + e_2) * (e_3 + e_4)$ is equivalent to $e_1 * e_3 + e_1 * e_4 + e_2 * e_3 + e_2 * e_4$.

Example:

```
tabular( X*F*(mean + sd) ~ 1 )
```

	F		All
X	a	mean	−0.04769
		sd	0.63181
	b	mean	0.01177
		sd	0.55410

2.2.3 $e_1 \sim e_2$

The tilde separates row specifications from column specifications, but otherwise acts the same as $*$, i.e. each row value applies to each column.

Example:

```
tabular( X*F ~ mean + sd )
```

	F	mean	sd
X	a	−0.04769	0.6318
	b	0.01177	0.5541

2.2.4 $e_1 = e_2$

The operator $=$ is used to set the name of e_2 to a displayed version of e_1. It is an abbreviation for `Heading(`e_1`)*`e_2. NB: because $=$ has lower operator precedence than any other operator, we usually put parentheses around these expressions, i.e. $(e_1 = e_2)$.

Example: `F` is renamed to `Newname`.

```
tabular( X*(Newname=F) ~ mean + sd )
```

	Newname	mean	sd
X	a	−0.04769	0.6318
	b	0.01177	0.5541

2.3 Terms in Formulas

R parses table formulas into sums, products, and bindings separated by the tilde formula operator. What comes between the operators are other expressions. Other than the pseudo-functions described in section 2.4, these are evaluated and the actions depend on the type of the resulting value.

2.3.1 Closures or other functions

If the expression evaluates to a function (e.g. it is the name of a function), then that function becomes the summary statistic to be displayed. The summary statistic should take a vector of values as input, and return a single value (either numeric, character, or some other simple printable value). If no summary function is specified, the default is `length`, to count the length of the vector being passed.

Note that only one summary function can be specified for any cell in the table or an error will be reported.

Example: `mean` and `sd` are specified functions; `n` is the renamed default statistic.

```
tabular( (F+1) ~ (n=1) + X*(mean + sd) )
```

		X	
F	n	mean	sd
a	5	−0.04769	0.6318
b	5	0.01177	0.5541
All	10	−0.01796	0.5611

2.3.2 Factors

If the expression evaluates to a factor, the dataset is broken up into subgroups according to the levels of the factor. Most of the examples above have shown this for the factor `F`, but this can also be used to display complete datasets:

Example: creating a factor to show all data. Use the `identity` function to display the values in each cell.

```
tabular( (i = factor(seq_along(X)))  ~
       Heading()*identity*(X+A +
              (F = as.character(F) ) ) )
```

i	X	A	F
1	−0.50219	a	a
2	0.13153	b	b
3	−0.07892	b	b
4	0.88678	a	a
5	0.11697	b	b
6	0.31863	a	a
7	−0.58179	a	a
8	0.71453	b	b
9	−0.82526	b	b
10	−0.35986	a	a

2.3.3 Logical vectors

If the expression evaluates to a logical vector, it is used to subset the data.

Example: creating subsets on the fly.

```
tabular( (X > 0) + (X < 0)  + 1
    ~ ((n = 1) + X*(mean + sd)) )
```

		X	
	n	mean	sd
X > 0	5	0.43369	0.3496
X < 0	5	−0.46960	0.2761
All	10	−0.01796	0.5611

2.3.4 Language Expressions

If the expression evaluates to a language object, e.g. the result of `quote()` or `substitute()`, then it will be replaced in the table formula by its result. This allows complicated table formulas to be saved and re-used. For examples, see section 2.5.

2.3.5 Other vectors

If the expression evaluates to something other than the above, then it is assumed to be a vector of values to be summarized in the table. If you would like to summarize a factor or logical vector, wrap it in `I()` to prevent special handling.

Note that the following must all be true, or an error will be reported:

- only one value vector can be specified for any cell in the table,
- all value vectors must be the same length,
- `is.atomic()` must evaluate to `TRUE` for the vector.

Example: treating a logical vector as values.

```
tabular( I(X > 0) + I(X < 0)
    ~ ((n=1) + mean + sd) )
```

	n	mean	sd
I(X > 0)	10	0.5	0.527
I(X < 0)	10	0.5	0.527

2.4 "Pseudo-functions"

Several directives to `tables` may be embedded in the table formula. This is done using "pseudo-functions". Syntactically they look like function calls, but reserved names are used. In most cases, their action applies to later factors in the term in which they appear. For example,

```
X*Justify(r)*(Y + Format(digits=2)*Z) + A
```

will apply the `Justify(r)` directive to both `Y` and `Z`, but the `Format(digits=2)` directive will only apply to `Z`, and neither will apply to `A`.

2.4.1 `Format()`

By default `tables` formats each column using the standard `format()` function, with arguments taken from the `format.tabular()` call (see section 2.1.2).

The `Format()` pseudo-function does two things: it changes the formatting, and it specifies that all values it applies to will be formatted together. The "call" to `Format` looks like a call to `format`, but without specifying the argument `x`. When `tabular()` formats the output it will construct `x` from the entries in the table governed by the `Format()` specification.

Example: The mean and standard deviation are both governed by the same format, so they are displayed with the same number of decimal places, chosen so that the smallest values (the means) show two significant digits.

```
tabular( (F+1) ~ (n=1)
           + Format(digits=2)*X*(mean + sd) )
```

		X	
F	n	mean	sd
a	5	−0.048	0.632
b	5	0.012	0.554
All	10	−0.018	0.561

For customized formatting, an alternate syntax is to pass a function call to `Format()`, rather than a list of arguments. The function should accept an argument named `x` (but as with the regular formatting, `x` should not be included in the formula), to contain the data. It should return a character vector of the same length as x.

Example: Use a custom function and `sprintf()` to display a standard error in parentheses.

```
stderr <- function(x) sd(x)/sqrt(length(x))
fmt <- function(x, digits, ...) {
  s <- format(x, digits=digits, ...)
  is_stderr <- (1:length(s)) > length(s) %/% 2
  s[is_stderr] <- sprintf("$(%s)$", s[is_stderr])
  s[!is_stderr] <- latexNumeric(s[!is_stderr])
  s
}
tabular( Format(fmt(digits=1))*(F+1) ~ X*(mean + stderr) )
```

	X	
F	mean	stderr
a	−0.05	(0.28)
b	0.01	(0.25)
All	−0.02	(0.18)

Character values in cells in the table are handled specially; see section 3.1 below.

2.4.2 `.Format()`

The pseudo-function `.Format()` is mainly intended for internal use. It takes a single integer argument, saying that data governed by this call uses the same formatting as the format specification indicated by the integer. In this way entries can be commonly formatted even when they are not contiguous. The integers are assigned sequentially as the format specification is parsed; users will likely need trial and error to find the right value in a complicated table with multiple formats.

Example: Format two separated columns with the same format.

```
tabular( (F+1) ~ X*(Format(digits=2)*mean
                    + (n=1) + .Format(1)*sd) )
```

	X		
F	mean	n	sd
a	−0.048	5	0.632
b	0.012	5	0.554
All	−0.018	10	0.561

2.4.3 `Heading()`

Normally `tabular()` generates row and column labels by deparsing the expression being tabulated. These can be changed by using the `Heading()` pseudo-function, which replaces the heading on the next object found. The heading can either be a name or a string in quotes. If the `character.only` argument is `TRUE`,

the expression will be evaluated to a string which will be used as a heading. LaTeX codes which are not syntactically valid R can be used either in quoted strings or with `character.only = TRUE`.

If no argument is passed, the next label is suppressed.

There's an optional argument `override`, which must be either `TRUE` or `FALSE` if present. If it is `TRUE` (or not present), then the heading will override a previously specified heading. If `FALSE`, it will not. The latter seems likely only to be of use in automatically generated code, and is used in the automatically generated labels for factors.

Another optional argument is `nearData`. This is used only when two terms in a table are concatenated using `+`, and they don't have the same number of rows or columns. Under the default `TRUE` value, the smaller one is moved closer to the data in the table (i.e. to the right for row labels, down for column labels); if `FALSE`, it is moved in the opposite direction.

Example: Replace `F` with a Greek Φ, and suppress the label for `X`.

```
tabular( (Heading("$\\Phi$")*F+1) ~ (n=1)
          + Format(digits=2)*Heading()*X*(mean + sd) )
```

Φ	n	mean	sd
a	5	−0.048	0.632
b	5	0.012	0.554
All	10	−0.018	0.561

Example: Use `nearData = FALSE` to push a label away from the data:

```
tabular( X*F + Heading("near")*X
       + Heading("far", nearData = FALSE)*X ~ mean + sd )
```

	F	mean	sd
X	a	−0.04769	0.6318
	b	0.01177	0.5541
	near	−0.01796	0.5611
far		−0.01796	0.5611

2.4.4 `Justify()`

The `Justify()` pseudo-function is used to specify the text justification of the headers and data values in the table. If called with one argument, that value is used for both labels and data; if called with two arguments, the first is used for the labels, the second for the data. If no `Justify()` specification is given, the default passed to `format()`, `print()` or `toLatex()` will be used. Values may be specified without quotes if they are legal R names; quoted strings may also be used. (The latter is useful for LaTeX output, for example `Justify("r@{}")`, to suppress column spacing on the right.)

Example:

```
tabular( Justify(r)*(F+1) ~ Justify(c)*(n=1)
   + Justify(c,r)*Format(digits=2)*X*(mean + sd) )
```

		X	
F	n	mean	sd
a	5	−0.048	0.632
b	5	0.012	0.554
All	10	−0.018	0.561

2.4.5 `Percent()`

The `Percent()` pseudo-function is used to specify a statistic that depends on other values in the table. It has two optional arguments:

- `denom="all"`: This specifies how the denominator (argument `y` to `fn` below) is set. The most commonly used values are `"all"`, meaning all values are used, `"row"`, meaning only the values in the current row are used, `"col"`, meaning only the values in the current column are used.

The special syntax `Equal(...)` will record the expressions in `...`, and ignore any factor based subsetting if the factor does not appear among the expressions. Similarly `Unequal(...)` will use values which differ in any of the expressions in `...` from the values in the current cell. (In fact, the mechanism is more general. The expressions in `Equal(...)` or `Unequal(...)` are deparsed and treated as strings. Any logical vector elsewhere in the table may be labelled with a string using the `labelSubset` function and those labels will be respected. Unlabelled logical vectors in the table formula will always be used for subsetting.)

If a logical vector is given, it is used to select which values form the denominator. Anything else is just passed to `fn` as given. - `fn=percent` This is the function which actually does the computation. The default definition is `function(x, y) 100*length(x) /length(y)`, giving the percentage count, but any other two argument function could be used.

These two examples are different ways of producing the same table:

```
tabular( (Factor(gear, "Gears") + 1)
         *((n=1) + Percent()
           + (RowPct=Percent("row"))
           + (ColPct=Percent("col")))
         ~ (Factor(carb, "Carburetors") + 1)
         *Format(digits=1), data=mtcars )
```

```
tabular( (Factor(gear, "Gears") + 1)
         *((n=1) + Percent()
           + (RowPct=Percent(Equal(gear)))  # Equal, not "row"
           + (ColPct=Percent(Equal(carb)))) # Equal, not "col"
         ~ (Factor(carb, "Carburetors") + 1)
         *Format(digits=1), data=mtcars )
```

		Carburetors						
Gears		1	2	3	4	6	8	All
3	n	3	4	3	5	0	0	15
	Percent	9	12	9	16	0	0	47
	RowPct	20	27	20	33	0	0	100
	ColPct	43	40	100	50	0	0	47
4	n	4	4	0	4	0	0	12
	Percent	12	12	0	12	0	0	38
	RowPct	33	33	0	33	0	0	100
	ColPct	57	40	0	40	0	0	38
5	n	0	2	0	1	1	1	5
	Percent	0	6	0	3	3	3	16
	RowPct	0	40	0	20	20	20	100
	ColPct	0	20	0	10	100	100	16
All	n	7	10	3	10	1	1	32
	Percent	22	31	9	31	3	3	100
	RowPct	22	31	9	31	3	3	100
	ColPct	100	100	100	100	100	100	100

2.4.6 Arguments()

The Arguments() pseudo-function is an exception to the rule that pseudo-functions apply to later factors in the table. What it does is to specify (additional) arguments to the summary function (see section 2.3.1). For example, the weighted.mean() function takes two arguments: x and w. To use it in a table, you would specify the values to use as x via the usual mechanism for the analysis variable (section 2.3.5), and include a term Arguments(w=weights) either before or after it. The function will be called as weighted.mean(x[subset], w=weights[subset]), where subset is a logical vector indicating which rows of data belong in the current cell.

It is actually a little more complicated than as described above. The arguments to Arguments are evaluated in full, then only those which are length n are subsetted. And if no analysis variable has been specified, but Arguments() has been, then the function will be called without the x[subset] argument. Finally, the Arguments() entry will not create a heading.

For example:

```
# This is the example from the weighted.mean help page
wt <- c(5,  5,  4,  1)/15
x <- c(3.7,3.3,3.5,2.8)
gp <- c(1,1,2,2)
tabular( (Factor(gp) + 1)
                ~ weighted.mean*x*Arguments(w = wt) )
```

	weighted.mean
gp	x
1	3.500
2	3.360
All	3.453

The same table (without the x heading) can be produced using

```
tabular( (Factor(gp) + 1)
                ~ Arguments(x, w = wt)*weighted.mean )
```

The order of the weighted.mean and Arguments() factors makes no difference.

2.4.7 DropEmpty()

DropEmpty() indicates that cells (or whole rows or columns of the table) should be dropped if they contain no observations. This will prevent ugly results like NA or NaN from showing up in the table.

This pseudo-function takes two optional arguments, which (with default value c("row", "col", "cell")) and empty (with default value "").

If the which argument contains "row", then any row in the table in which all cells are empty will be dropped. Similarly, if it contains "col", empty columns will be dropped. If it contains "cell", then cells in rows and columns that are not dropped will be set to the empty string.

For example, without using DropEmpty(), this table is ugly:

```
set.seed(730)
df <- data.frame(Label = LETTERS[1:9],
         Group = rep(letters[1:3], each=3),
         Value = rnorm(9),
         stringsAsFactors = TRUE)
tabular( Label ~ Group*Value*mean,
        data = df[1:6,])
```

	Group		
Label	a Value mean	b Value mean	c Value mean
A	−0.92605	*NaN*	*NaN*
B	−0.69747	*NaN*	*NaN*
C	0.05293	*NaN*	*NaN*
D	*NaN*	0.7782	*NaN*
E	*NaN*	0.9822	*NaN*
F	*NaN*	−1.0628	*NaN*
G	*NaN*	*NaN*	*NaN*
H	*NaN*	*NaN*	*NaN*
I	*NaN*	*NaN*	*NaN*

This looks much better:

```
tabular( Label ~ Group*Value*mean*
            DropEmpty(empty="."),
         data = df[1:6,])
```

	Group	
Label	a Value mean	b Value mean
A	−0.92605	.
B	−0.69747	.
C	0.05293	.
D	.	0.7782
E	.	0.9822
F	.	−1.0628

2.5 Formula Functions

Currently several examples of formula functions are provided. Not all are particularly robust; e.g. `Hline()` only works for LaTeX output and must be in a particular position in the formula. Users can provide their own as well. Such functions should return a language object, which will be substituted into the formula in place of the formula function call.

2.5.1 `All()`

This function expands all the columns from a dataframe into separate variables in the table. It has syntax

```
All(df, numeric=TRUE, character=FALSE, logical=FALSE,
        factor=FALSE, complex=FALSE, raw=FALSE, other=FALSE,
        texify=getOption("tables.texify", FALSE))
```

The arguments are

- `df`: A dataframe or matrix whose columns are to be displayed
- `numeric`, `character`, `logical`, `factor`, `complex` and `raw`: Whether to include columns of the corresponding types in the table.
- `other`: Whether to include columns that match none of the previous types.
- `texify`: Whether to escape LaTeX special characters. See section 3.1.

If functions are given for any of the selection arguments, the columns will be transformed according to

the specified function before inclusion. For example, using `factor=as.character` will convert factors into character vectors in the table.

Example: Show the means of the numeric columns in the iris data.

```
tabular( Species ~ Heading()*mean*All(iris), data=iris)
```

Species	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width
setosa	5.006	3.428	1.462	0.246
versicolor	5.936	2.770	4.260	1.326
virginica	6.588	2.974	5.552	2.026

2.5.2 `AllObs()`, `RowNum()`

The `AllObs()` function displays all of the observations in a dataset. It does this by creating a factor with a different level for each observation, and a summary statistic function which just displays the observation. It works with `DropEmpty()` to drop rows (or columns) from the table if they correspond to non-existent observations. For example,

```
df <- mtcars[1:10,]
tabular(Factor(cyl)*Factor(gear)*AllObs(df) ~
                rownames(df) + mpg, data=df)
```

cyl	gear	rownames(df)	mpg
4	4	Datsun 710	22.8
		Merc 240D	24.4
		Merc 230	22.8
6	3	Hornet 4 Drive	21.4
		Valiant	18.1
	4	Mazda RX4	21.0
		Mazda RX4 Wag	21.0
		Merc 280	19.2
8	3	Hornet Sportabout	18.7
		Duster 360	14.3

Often (as with the `mtcars` dataset) the full dataset takes a lot of space to display. In that case, it can be displayed in multiple columns using a combination of the `AllObs()` and `RowNum()` functions. Because this affects both rows and columns in the resulting table, the code is a little unusual. You would normally compute the `RowNum()` formula function outside the call to `tabular()`, and include it in the row specification wrapped in `I()` and in the column specification in the `within` argument to `AllObs()`. For example,

```
rownum <- with(mtcars, RowNum(list(cyl, gear)))
tabular(Factor(cyl)*Factor(gear)*I(rownum) ~
        mpg * AllObs(mtcars, within = list(cyl, gear, rownum)),
        data=mtcars)
```

cyl	gear			mpg		
4	3	21.5				
	4	22.8	24.4	22.8	32.4	30.4
		33.9	27.3	21.4		
	5	26.0	30.4			
6	3	21.4	18.1			
	4	21.0	21.0	19.2	17.8	
	5	19.7				
8	3	18.7	14.3	16.4	17.3	15.2
		10.4	10.4	14.7	15.5	15.2
		13.3	19.2			
	5	15.8	15.0			

Despite its name, `RowNum` can be used to specify columns instead of rows, for a column-major display. In this case, its `perrow` argument should be interpreted as "per column". For example,

```
rownum <- with(mtcars, RowNum(list(cyl, gear), perrow = 2))
tabular(Factor(cyl)*Factor(gear)*
           AllObs(mtcars, within = list(cyl, gear, rownum)) ~
               mpg * I(rownum),
        data=mtcars)
```

cyl	gear			mpg			
4	3	21.5					
	4	22.8	22.8	30.4	27.3		
		24.4	32.4	33.9	21.4		
	5	26.0					
		30.4					
6	3	21.4					
		18.1					
	4	21.0	19.2				
		21.0	17.8				
	5	19.7					
8	3	18.7	16.4	15.2	10.4	15.5	13.3
		14.3	17.3	10.4	14.7	15.2	19.2
	5	15.8					
		15.0					

2.5.3 `Hline()`

This function produces horizontal lines in the table. It only works for LaTeX output, and must be the first factor in a term in the table formula. It has syntax

```
Hline(columns)
```

The argument is

- '`columns`: An optional vector listing which columns should get the line.

Example:

```
tabular( Species + Hline(2:5) + 1
                          ~ Heading()*mean*All(iris), data=iris)
```

Species	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width
setosa	5.006	3.428	1.462	0.246
versicolor	5.936	2.770	4.260	1.326
virginica	6.588	2.974	5.552	2.026
All	5.843	3.057	3.758	1.199

2.5.4 `Literal()`

This function inserts literal text as a label. It has syntax

```
Literal(x)
```

The single argument is the text to insert. It is used by the `Hline()` function to insert the text.

2.5.5 `PlusMinus()`

This function produces table entries like $x \pm y$ with an optional header. It has syntax

```
PlusMinus(x, y, head, xhead, yhead, digits=2, ...)
```

The arguments are

- '`x, y`: These are expressions which should each generate a single column in the table. The `x` value will be flush right, the `y` value will be flush left, with the ± symbol between.
- '`head`: If not missing, this header will be put over the pair of columns.
- '`xhead, yhead`: If not missing, these will be put over the individual columns.
- '`digits, ...`: These arguments will be passed to the standard `format()` function.

Example: Display mean ± standard error.

```
stderr <- function(x) sd(x)/sqrt(length(x))
tabular( (Species+1) ~ All(iris)*
          PlusMinus(mean, stderr, digits=1), data=iris )
```

Species	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width
setosa	5.01 ± 0.05	3.43 ± 0.05	1.46 ± 0.02	0.25 ± 0.01
versicolor	5.94 ± 0.07	2.77 ± 0.04	4.26 ± 0.07	1.33 ± 0.03
virginica	6.59 ± 0.09	2.97 ± 0.05	5.55 ± 0.08	2.03 ± 0.04
All	5.84 ± 0.07	3.06 ± 0.04	3.76 ± 0.14	1.20 ± 0.06

2.5.6 `Paste()`

This function produces table entries made up of multiple values. It has syntax

```
Paste(..., head, digits=2, justify="c", prefix="", sep="",
      postfix="")
```

The arguments are

- '`...`: Expressions to be displayed in the columns of the table.
- '`head`: If not missing, this will be used as a column heading for the combined columns.
- '`digits`: Digits used in formatting. If a single value is given, all columns will be formatted in common. If multiple values are given, each is formatted separately.
- '`justify`: One or more justifications to use on the individual columns.
- '`prefix, sep, postfix`: Text to use before, between, and after the columns.

Example: Display a confidence interval.

```
lcl <- function(x) mean(x) - qt(0.975, df=length(x)-1)*stderr(x)
ucl <- function(x) mean(x) + qt(0.975, df=length(x)-1)*stderr(x)
tabular( (Species+1) ~ All(iris)*
          Paste(lcl, ucl, digits=2,
                head="95\\% CI", sep=",", prefix="[",
                postfix="]"),
          data=iris )
```

	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width
Species	95% CI	95% CI	95% CI	95% CI
setosa	[4.91, 5.11]	[3.32, 3.54]	[1.41, 1.51]	[0.22, 0.28]
versicolor	[5.79, 6.08]	[2.68, 2.86]	[4.13, 4.39]	[1.27, 1.38]
virginica	[6.41, 6.77]	[2.88, 3.07]	[5.40, 5.71]	[1.95, 2.10]
All	[5.71, 5.98]	[2.99, 3.13]	[3.47, 4.04]	[1.08, 1.32]

2.5.7 `Factor()`, `RowFactor()` and `Multicolumn()`

The `Factor()` function converts its argument into a factor, but keeps the original name for a column heading. `RowFactor()` is designed to be used only for LaTeX output: it produces multiple rows the way a factor does, but with more flexibility in the formatting. The `Multicolumn()` function is also designed for LaTeX output: it displays factor levels in the style where the level is displayed across multiple columns on its own line.

They have syntax

```
Factor(x, name, levelnames, texify=getOption("tables.texify", FALSE))
RowFactor(x, name, levelnames, spacing=3, space=1,
                    nopagebreak="\\nopagebreak", texify=getOption("tables.texify", FALSE))
Multicolumn(x, name, levelnames, width=2, first=1, justify="l",
                    texify=getOption("tables.texify", FALSE))
```

The arguments are

- `x`: A variable to be treated as a factor.
- `name`: The name to be used for the factor; by default, the name passed as `x`.
- `levelnames` An optional argument to allow customization of the displayed level names.
- `texify` Whether to escape LaTeX special characters. See section 3.1.
- `spacing` Extra spacing is added before every group of `spacing` lines.
- `space` How much extra space to add (in "ex" units).
- `nopagebreak` Macro to insert to suppress page breaks except between groups.
- `width` How many columns for the label?
- `first`: What is the first column?
- `justify`: What justification to use.

Example: Show the first 15 lines of the iris dataset, in groups of 5 lines.

```
subset <- 1:15
tabular( RowFactor(subset, "$i$", spacing=5)  ~
       All(iris[subset,], factor=as.character)*Heading()*identity )
```

i	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
1	5.1	3.5	1.4	0.2	setosa
2	4.9	3.0	1.4	0.2	setosa
3	4.7	3.2	1.3	0.2	setosa
4	4.6	3.1	1.5	0.2	setosa
5	5.0	3.6	1.4	0.2	setosa
6	5.4	3.9	1.7	0.4	setosa
7	4.6	3.4	1.4	0.3	setosa
8	5.0	3.4	1.5	0.2	setosa
9	4.4	2.9	1.4	0.2	setosa
10	4.9	3.1	1.5	0.1	setosa
11	5.4	3.7	1.5	0.2	setosa
12	4.8	3.4	1.6	0.2	setosa
13	4.8	3.0	1.4	0.1	setosa
14	4.3	3.0	1.1	0.1	setosa
15	5.8	4.0	1.2	0.2	setosa

To add extra space after each high level group in a multi-way classification, use `spacing = 1`. For example:

```
set.seed(1000)
dat <- expand.grid(Block=1:3, Treatment=LETTERS[1:2],
                                  Subset=letters[1:2])
dat$Response <- rnorm(12)
toLatex( tabular( RowFactor(Block, spacing=1)
                * RowFactor(Treatment, spacing=1, space=0.5)
                * Factor(Subset)
                ~ Response*Heading()*identity, data=dat),
                options=list(rowlabeljustification="c") )
```

Block	Treatment	Subset	Response
1	A	a	−0.44578
		b	−0.47587
	B	a	0.63939
		b	−1.37312
2	A	a	−1.20586
		b	0.71975
	B	a	−0.78655
		b	−0.98243
3	A	a	0.04113
		b	−0.01851
	B	a	−0.38549
		b	−0.55449

For longer tables, the `"longtable"` environment allows the table to cross page boundaries. Using this is more complicated, as in the example below. The `toprule` setting inserts the caption as well as the top rule, because the `longtable` package requires it to be *within* the table. The `midrule` setting gets the headings to repeat on subsequent pages. (I've done all of this in a way that is compatible with the `booktabs` style; if you want the default style, use `\hline` in place of the `booktabs` `\toprule` and `\midrule` macros in the `options` settings instead.) To avoid extra spacing at the top of those pages, we need to undo the automatic addition

of a \normalbaselineskip there, and use suppressfirst=FALSE so that the first page doesn't get messed up. Whew!

```
subset <- 1:50
toLatex( tabular( RowFactor(subset, "$i$", spacing=5,
                                              suppressfirst=FALSE)  ~
       All(iris[subset,], factor=as.character)*Heading()*identity ),
       options = list(tabular="longtable",
          toprule="\\caption{This table crosses page boundaries.}\\\\
              \\toprule",
midrule="\\midrule\\\\[-2\\normalbaselineskip]\\endhead\\hline\\endfoot") )
```

Table 1: This table crosses page boundaries.

i	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
1	5.1	3.5	1.4	0.2	setosa
2	4.9	3.0	1.4	0.2	setosa
3	4.7	3.2	1.3	0.2	setosa
4	4.6	3.1	1.5	0.2	setosa
5	5.0	3.6	1.4	0.2	setosa
6	5.4	3.9	1.7	0.4	setosa
7	4.6	3.4	1.4	0.3	setosa
8	5.0	3.4	1.5	0.2	setosa
9	4.4	2.9	1.4	0.2	setosa
10	4.9	3.1	1.5	0.1	setosa
11	5.4	3.7	1.5	0.2	setosa
12	4.8	3.4	1.6	0.2	setosa
13	4.8	3.0	1.4	0.1	setosa
14	4.3	3.0	1.1	0.1	setosa
15	5.8	4.0	1.2	0.2	setosa
16	5.7	4.4	1.5	0.4	setosa
17	5.4	3.9	1.3	0.4	setosa
18	5.1	3.5	1.4	0.3	setosa
19	5.7	3.8	1.7	0.3	setosa
20	5.1	3.8	1.5	0.3	setosa
21	5.4	3.4	1.7	0.2	setosa
22	5.1	3.7	1.5	0.4	setosa
23	4.6	3.6	1.0	0.2	setosa
24	5.1	3.3	1.7	0.5	setosa
25	4.8	3.4	1.9	0.2	setosa
26	5.0	3.0	1.6	0.2	setosa
27	5.0	3.4	1.6	0.4	setosa
28	5.2	3.5	1.5	0.2	setosa
29	5.2	3.4	1.4	0.2	setosa
30	4.7	3.2	1.6	0.2	setosa

Table 1: This table crosses page boundaries.

i	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
31	4.8	3.1	1.6	0.2	setosa
32	5.4	3.4	1.5	0.4	setosa
33	5.2	4.1	1.5	0.1	setosa
34	5.5	4.2	1.4	0.2	setosa
35	4.9	3.1	1.5	0.2	setosa
36	5.0	3.2	1.2	0.2	setosa
37	5.5	3.5	1.3	0.2	setosa
38	4.9	3.6	1.4	0.1	setosa
39	4.4	3.0	1.3	0.2	setosa
40	5.1	3.4	1.5	0.2	setosa
41	5.0	3.5	1.3	0.3	setosa
42	4.5	2.3	1.3	0.3	setosa
43	4.4	3.2	1.3	0.2	setosa
44	5.0	3.5	1.6	0.6	setosa
45	5.1	3.8	1.9	0.4	setosa
46	4.8	3.0	1.4	0.3	setosa
47	5.1	3.8	1.6	0.2	setosa
48	4.6	3.2	1.4	0.2	setosa
49	5.3	3.7	1.5	0.2	setosa
50	5.0	3.3	1.4	0.2	setosa

To suppress the row numbering, use `suppress=3` in the call to tabular. (It is 3 because we need to suppress the column heading, the rewritten labels for the rows, and the original labels. Trial and error is the best way to determine this!) Unfortunately, the spacing features of `RowFactor()` won't work without the row labels.

```
subset <- 1:10
tabular( Factor(subset)  ~
       All(iris[subset,], factor=as.character)*Heading()*identity,
       suppress=3 )
```

Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
5.1	3.5	1.4	0.2	setosa
4.9	3.0	1.4	0.2	setosa
4.7	3.2	1.3	0.2	setosa
4.6	3.1	1.5	0.2	setosa
5.0	3.6	1.4	0.2	setosa
5.4	3.9	1.7	0.4	setosa
4.6	3.4	1.4	0.3	setosa
5.0	3.4	1.5	0.2	setosa
4.4	2.9	1.4	0.2	setosa
4.9	3.1	1.5	0.1	setosa

(It is actually possible to get this to work with `RowFactor()`, but it is ugly: set the name and level names to `""`, and set the justification to `"l@{}"` to suppress the intercolumn spacing. Then the column of row labels will be there, but it will be zero width and invisible.)

`RowFactor` with `spacing > 1` will add the `nopagebreak` macro at the beginning of each label except the first in the group. This can produce LaTeX errors in any column except the first one. One workaround for

this is to post-process the table to move the macro. For example, if `tab` contains the result of `tabular()` and LaTeX complains about misplaced `\nopagebreak` macros, this will allow it to be displayed properly:

```
code <- capture.output( toLatex( tab ) )
code <- sub("^(.*)(\\\\nopagebreak )", "\\2\\1", code)
cat(code, sep = "\n")
```

To get group labels to span multiple columns, the `levelnames` argument can be used with embedded LaTeX code. For example,

```
tabular( Multicolumn(Species, width=3,
            levelnames=paste("\\textit{Iris", levels(Species),"}"))
            * (mean + sd)  ~ All(iris), data=iris, suppress=1)
```

	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width
Iris setosa				
mean	5.0060	3.4280	1.4620	0.2460
sd	0.3525	0.3791	0.1737	0.1054
Iris versicolor				
mean	5.9360	2.7700	4.2600	1.3260
sd	0.5162	0.3138	0.4699	0.1978
Iris virginica				
mean	6.5880	2.9740	5.5520	2.0260
sd	0.6359	0.3225	0.5519	0.2747

3 Further Details

3.1 Formatting

As mentioned in 2.4.1, formatting in `tables` depends on the standard `format()` function or other user-selected functions. Here are the details of how it is done.

The `format.tabular()` method does the first part of the work. First, it constructs the calls to the appropriate formatting functions, and uses them to format all of the non-character entries in the table. The character entries are left as-is, except as described below. This converts the `tabular` object to a character array.

The procedure goes as follows:

1. Entries in the table without specified formatting are formatted first, separately by column using the `format()` function. This is so that entries in a given column will end up with the same character width and (with the default settings) with the same number of decimal places.
2. Entries in the table with specified formatting are grouped according to the format specification. For example, if two columns both share the same `Format()`, they will be formatted in a single call. This results in such entries ending up with the same character width and (with the default settings) with the same number of decimal places.
3. If the `toLatex` argument is `TRUE`, any numeric entries are passed to the `latexNumeric()` function (see 2.1.6), which replaces blanks and minus signs with fixed width spaces and LaTeX minus signs so that all entries will display in the same width. This means that numeric values will normally have decimal points aligned, unless the formatting function explicitly removes leading spaces. Non-numeric entries are passed through the `Hmisc::latexTranslate` function so that special characters are displayed properly.
4. If the `toLatex` argument is `FALSE`, an attempt is made to justify the results using simple ASCII spacing, according to the `Justify()` specification with the `justification` argument used as a default.

Note that LaTeX special characters will not be escaped in data when `toLatex()` is called, but row and column headings generated by `All()`, `Factor()`, etc. will by default not have the escapes done. Those functions

have a `texify` argument that can be set to `TRUE` to enable this behaviour (e.g. if the label is not meant to be processed by LaTeX). For example, with the definition

```
df <- data.frame(A = factor(c( "$", "\\" ) ), B_label=1:2)
```

the code

```
tabular( mean ~ A*B_label, data=df )
```

would fail, as the labels would include the special characters. But this will work, provided the `Hmisc` package is available:

```
options(tables.texify = TRUE)
tabular( mean ~ Factor(A)*All(df), data=df )
```

```
## Loading required namespace: Hmisc
```

	A	
	$	\
	B_label	B_label
mean	1	2

Use of the `texify` option requires that the suggested package `"Hmisc"` be available.

As mentioned above, character values in cells in the table are handled specially. If the default `format` function (or a custom function named `format`) is used, then those character values are not formatted, they are just copied into the result. (This is so that a column can have mixed numeric and character values, and the numerics are not converted to character before formatting.) If you want to use `format` on character values, you will need to use a custom formatting function with a different name.

3.2 Missing Values

By default, most summary statistics in R return `NA` if any of the input values are `NA`, but have ways to treat `NA` differently. For example, the `mean()` function has the `na.rm` argument:

```
dat <- data.frame( a = c(1, 2, 3, NA), b = 1:4 )
mean(dat$a)
```

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

```
mean(dat$a, na.rm=TRUE)
```

```
## [1] 2
```

The `tabular()` function itself has no way to specify special `NA` handling, but there are several ways to do this yourself, depending on how you want them handled. To ignore `NA` values within the column, define a new function which sets the different behaviour. For example,

```
Mean <- function(x) base::mean(x, na.rm=TRUE)
tabular( Mean ~ a + b, data=dat )
```

	a	b
Mean	2	2.5

An alternative approach is to use `na.omit()` to work on a subset of your data which has rows with any missing values removed, e.g.

```
tabular( mean ~ a + b, data = na.omit(dat) )
```

	a	b
mean	2	2

A third possibility is to use the `complete.cases()` function to remove missings only from some columns, e.g.

```
tabular(
  Mean ~ (1 + Heading(Complete)*complete.cases(dat)) * (a + b),
                data=dat )
```

	All		Complete	
	a	b	a	b
Mean	2	2.5	2	2

Missing values in factors are normally ignored, i.e. observations whose value is missing won't match any category. If you would like `NA` to be used as an additional category, use `exclude = NULL` in a call to `factor()` when you create the variable, e.g. compare the following two tables:

```
A <- factor(dat$a)
tabular( A + 1 ~ (n=1))
```

A	n
1	1
2	1
3	1
All	4

```
A <- factor(dat$a, exclude = NULL)
tabular( A + 1 ~ (n=1) )
```

A	n
1	1
2	1
3	1
NA	1
All	4

3.3 Subsetting and Joining Tables

It is possible to select a subset of a table using the usual R matrix indexing on the table object. For example, this table contains rows with no data in them, and those yield ugly NA and NaN statistics:

```
set.seed(1206)
q <- data.frame(p = rep(c("A","B"),each=10,len=30),
                           a = rep(c(1,2,3),each=10),id=seq(30),
                           b = round(runif(30,10,20)),
                           c = round(runif(30,40,70)),
        stringsAsFactors = FALSE)
tab <- tabular((Factor(p)*Factor(a)+1)
                ~ (N = 1) + (b + c)*(mean+sd), data = q)
tab
```

			b		c	
p	a	N	mean	sd	mean	sd
A	1	10	14.40	3.026	55.70	6.447
	2	0	*NaN*	*NA*	*NaN*	*NA*
	3	10	14.50	2.877	52.80	8.954
B	1	0	*NaN*	*NA*	*NaN*	*NA*
	2	10	14.40	2.836	56.30	7.889
	3	0	*NaN*	*NA*	*NaN*	*NA*
	All	30	14.43	2.812	54.93	7.714

To omit those rows, use matrix-like subsetting to select the rows where the first column of data (i.e. N) is greater than zero:

```
tab[ tab[,1] > 0, ]
```

			b		c	
p	a	N	mean	sd	mean	sd
A	1	10	14.40	3.026	55.70	6.447
	3	10	14.50	2.877	52.80	8.954
B	2	10	14.40	2.836	56.30	7.889
	All	30	14.43	2.812	54.93	7.714

Similarly, `cbind()` can be used to join tables that have identical row labels, and `rbind()` can be used to join tables with identical column labels. Thus the top part of the table above could be produced in another way:

```
formula <- Factor(p)*Factor(a) ~
       (N = 1) + (b + c)*(mean+sd)
tab <- NULL
for (sub in c("A", "B"))
    tab <- rbind(tab, tabular( formula,
                               data = subset(q, p == sub) ) )
tab
```

			b		c	
	a	N	mean	sd	mean	sd
A	1	10	14.4	3.026	55.7	6.447
	3	10	14.5	2.877	52.8	8.954
B	2	10	14.4	2.836	56.3	7.889

It is also possible to edit the row or column labels after constructing the table. For example,

```
colLabels(tab)
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]      b    <NA> c    <NA>
## [2,] N    mean sd   mean sd
## Attributes:  dim, justification, colnamejust, justify, suppress, nearData, class
```

```
labs <- colLabels(tab)
labs[1, 2] <- "New label"
colLabels(tab) <- labs
```

```
tab
```

			New label		c	
	a	N	mean	sd	mean	sd
A	1	10	14.4	3.026	55.7	6.447
	3	10	14.5	2.877	52.8	8.954
B	2	10	14.4	2.836	56.3	7.889

Note that `<NA>` in the column labels means "same as the label to the left", and in the row labels it means "same as the label above". This is used in constructing multi-column or multi-row labels.

3.4 `knitr`, `rmarkdown` and `kableExtra` support

This vignette was originally written many years ago using `Sweave`, and is still available in that format. Nowadays I would recommend most users to use `knitr` instead: it is easier and more flexible. The input may be in Noweb syntax very similar to `Sweave`, or Markdown syntax using the `rmarkdown` package, as in this file.

One specific advantage of using `knitr` or `rmarkdown` is that explicit calls to `toLatex()` are not needed: by default, tabular objects will print in the appropriate formatting for LaTeX or HTML output.

The `kableExtra` package may be used to customize displays. For example, the code below causes the table to be full width, and the colour of the 4th column is changed. These features require additional LaTeX packages; see the `kableExtra` documentation for details.

```
library(magrittr)
library(kableExtra)
toKable(tab) %>%
  kable_styling(full_width = TRUE) %>%
  column_spec(4, color = "red")
```

			New label		c	
	a	N	mean	sd	mean	sd
A	1	10	14.4	3.026	55.7	6.447
	3	10	14.5	2.877	52.8	8.954
B	2	10	14.4	2.836	56.3	7.889

See the HTML vignette (which is written in `rmarkdown`) for more discussion and examples.

3.5 Captions, labels, etc.

LaTeX breaks the description of tables into two parts: the `tabular` environment holding the data, and the optional `table` environment surrounding it, where captions, labels, where to place the table in the document, etc. are all specified. The `tables` package concentrates on the details of the `tabular` part, because I didn't want to duplicate the myriad options in LaTeX to set up the `table` wrapper. However, others are not so lazy, and Yihui Xie's `knitr` package includes the `kable()` function which does these things. (It is much less flexible about the actual contents, however.) Rather than copying all his code, I have added the `latexTable` function. It uses `kable()` to produce a dummy table, then replaces the `tabular` part with the result of the `tabular()` function from this package. For example, this code produces Table 2:

```
latexTable(tabular((Species + 1) ~ (n=1) + Format(digits=2)*
                   (Sepal.Length + Sepal.Width)*(mean + sd),
                   data=iris),
           caption = "Iris sepal data", label = "sepals")
```

which should have floated to the top or bottom of page 29.

Table 2: Iris sepal data

		Sepal.Length		Sepal.Width	
Species	n	mean	sd	mean	sd
setosa	50	5.01	0.35	3.43	0.38
versicolor	50	5.94	0.52	2.77	0.31
virginica	50	6.59	0.64	2.97	0.32
All	150	5.84	0.83	3.06	0.44

4 Acknowledgments

I gratefully acknowledge helpful suggestions and hints from Rich Heiberger, Frank Harrell, Dieter Menne, Marius Hofert, Jeff Newmiller and Jeffrey Miller. Hao Zhu was extremely helpful in adding the `kableExtra` support.

References

Fear, Simon. 2005. *Publication Quality Tables in Latex.* http://www.ctan.org/tex-archive/macros/latex/contrib/booktabs.