Journal of Statistical Software

MMMMMM YYYY, Volume VV, Issue II. *doi: 10.18637/jss.v000.i00*

cleanR: Maid for Cleaning Datasets in R

Anne H. Petersen
Biostatistics
Department of Public Health
University of Copenhagen

Claus Thorn Ekstrøm
Biostatistics
Department of Public Health
University of Copenhagen

Abstract

Data cleaning and -validation are the first steps in any data analysis, as the validity of the conclusions from the analysis are hinged on the quality of the input data. Mistakes in the data can arise for any number of reasons, including erroneous codings, malfunctioning measurement equipment, inconsistent data generation manuals and many more. Ideally, a human investigator should go through each variable in the dataset and look for potential errors — both in input values and coding. segway

We describe an R package which implements an extensive and customizeable suite of checks to be applied to the variables in a dataset in order to identify potential problems in the corresponding variables. The results can be presented in an auto-generated, non-technical, stand-alone overview document, intended to be perused by an investigator with an understanding of the variables in the data and the experimental design, but not necessarily knowledge of `R`. Thereby, `cleanR` aids the dialogue between data analysist and field experts, while also providing easy documentation of data cleaning steps and data quality control. Moreover, the `cleanR` solution changes the data cleaning process from the usual ad hoc approach to a systematic, well-documented endeavor. `cleanR` also provides a suite of more typical `R` tools for interactive data quality assessment and -cleaning.

The **cleanR** package is designed to be easily extended with custom user-created checks that are relevant in particular situations.

Keywords: data cleaning, quality control, R.

1. Introduction

Statisticians and data analysts spend a large portion of their time on data cleaning and on data wrangling. Packages such as **data.table**, and **plyr** have made data wrangling a lot easier in R, but there are only a few tools available for data cleaning. What are these tools? More here about the solutions already available.

A text here (with offset in what tools are already available) that

- Makes the point that data cleaning is usually done in a very ad hoc manner
- Maybe hints to the discussion later about poor documentation of data cleaning/quality assessment as the status quo for most analysts
- Gives a specification of what we mean by data errors/mistakes/whatever. I'm not completely sure this will be clear to everyone.

Data cleaning is a time consuming endeavor, as it inherently requires human interaction since every dataset is different and the variables in the dataset can only be understood in the proper context of their origin. Segway: Something like maybe this is why proper data cleaning is not always done... In many situations these errors are discovered in the process of the data analysis (e.g., a categorical variable with numeric labels is wrongly classified as a numeric variable[1] or a variable where all values have erroneously been coded to the same value), but in other cases a human with knowledge about the data context area is needed to identify possible mistakes in the data (e.g., if there are 4 categories for a variable that should only have 3). segway

The `cleanR` approach to data cleaning and -quality assessment is characterized by two principles. First of all, there is no need for data cleaning to be an ad hoc procedure. Often, we have a very clear idea of what flags are raisable in a given dataset before we look at it, as we were the ones to produce it in the first place. This means that data cleaning can easily be a well-documented, well-specified procedure. This angle on data cleaning also adds to the transparency of this first step of data analysis: Though data cleaning might be regarded as a somewhat dull activity, adequate data cleaning is crucial in any data analysis. In order to aid this principle, `cleanR` provides easy-to-use, automated tools for data quality assessment in `R` on which data cleaning decisions can be build. This quality assessment is presented in a auto-generated overview document, readable by data analysts and field experts alike, thereby also contributing to a inter-field dialogue about the data at hand. Oftentimes, e.g. separating faulty codings of a numeric value from unusual, but correct, values requires problem-specific expertise that might not be held by the data analyst. Hopefully, having easy access to data descriptions through `cleanR` will help this necessary knowledge sharing.

While `cleanR` is build for auto-generating data quality assessment overview documents, we also wish to emphasize that it is *not* a tool for automatic data cleaning. This qualifies as the second principle of `cleanR`: Data cleaning decisions should always be made by humans. Therefore, `cleanR` does not provide any tools for "fixing" errors in the data. However, we do provide interactive functions that can be used to identify potentially erroneous entries in a dataset and that can make it easier to solve data issues.

This manuscript is structured as follows: First, we introduce the representative of the first principle, namely the `clean()` function, which generates data cleaning overview documents. In the `cleanR` package, we have provided a number of default cleaning steps that cover the data cleaning challenges, we find to be most common. However, every dataset is different, and some datasets might include problems that cannot be detected by our data checking functions. In Section 3 proper reference here, we turn to the question of how such `cleanR`

[1] Is that something we can fix in clean()?

extensions can be made such that they are integrable with the `clean()` function and with other tools available in `cleanR`. In this section, we also present the interactive mode of `cleanR`, as motivated by the second principle above. At last, in Section 4 proper ref, we discuss a number of examples of specific data cleaning challenges and how `cleanR` can be used to solve them.

2. Something like "producing a data cleaning overview"

In **cleanR**, the `clean` function is the primary workhorse and knowledge of nothing more than this function is required to produce data cleaning outputs. The data cleaning output itself is an overview document, intended for reading by humans, in either pdf or html format. Appendix something provides an example of a data cleaning output document, produced by calling `clean` on the dataset `toyData` available in `cleanR`. The first two pages of this data cleaning output are also available in Figure 2. `toyData` is a very small (15 by 6), artificial dataset, whose only purpose is to illustrate the main capabilities of `cleanR`. The following commands load the dataset and produce the cleaning output:

```
> library(cleanR)
> data(toyData)
> toydata
```

```
   var1 var2  var3        var4 var5       var6
1   red    1     a -0.65959383    1 Irrelevant
2   red    1     a  0.08671649    2 Irrelevant
3   red    1     a -0.10951326    3 Irrelevant
4   red    2     a  0.08630221    4 Irrelevant
5   red    2     a -1.84311184    5 Irrelevant
6   red    6     b  0.92210680    6 Irrelevant
7   red    6     b  1.01921086    7 Irrelevant
8   red    6     b -0.92428326    8 Irrelevant
9   red  999     c -0.65340163    9 Irrelevant
10  red   NA     c  0.21133941   10 Irrelevant
11 blue    4     c  0.91783009   11 Irrelevant
12 blue   82     .  0.10313983   12 Irrelevant
13 blue   NA        0.16954218   13 Irrelevant
14 <NA>  NaN other  0.41967230   14 Irrelevant
15 <NA>    5 OTHER  0.77143836   15 Irrelevant
```

```
> clean(toyData)
```

By default, a pdf overview document is produced, saved to the disc and opened for inspection. Turning to Figure 2, we see that such a data cleaning output document consists of two parts. First, an overview of what was done is presented under the title *Data cleaning summary*. Secondly, each variable in the dataset is presented in turn using (up to) three tools in the *Variable list*: A table summarizing key features of the variable, a figure visualizing its distribution and potentially also a list of flagged issues. For instance, in the `numeric`-type

Argument	Description	Default value
Control ...eh?		
`useVar`	What variables should be used?	`NULL` (corresponding to all variables)
`ordering`	Ordering of the variables in the data summary (as is or alphabetical)	`"asIs"`
`onlyProblematic`	Should only variables flagged as problematic be included in the variable list?	`FALSE`
`listChecks`	Should an overview of what checks were performed by listed in the *Data cleaning summary*?	`TRUE`
Control SVC-step		
`mode`	What steps should be performed for each variable (out of the three possibilities *summarize*, *visualize*, *check*)?	`c("summarize", "visualize", "check")`
`labelled_as`	How should variables of class `labelled` be handled (as factors, is missing values or by ignoring labels)?	`"factor"`
`smartNum`	Should numerical values with only a few unique levels be flagged an treated as factor variables?	`TRUE`
`maxProbVals`	Maximum number of problematic values to print, if any are found in data checks	`Inf`
`maxDecimals`	Maximum number of decimals to print for numeric values in the variable list	`2`
`twoCol`	Should the summary table and visualizations be placed side-by-side (in two columns)?	`TRUE`
Control output		
`output`	Type of output file to be produced (html, pdf or ?? what does screen mean here??)	`"pdf"`
`render`	Should the output file be rendered from markdown?	`TRUE`
`openResult`	If a pdf/html file is rendered, should it automatically open afterwards?	`TRUE`

Table 1: A selection of commonly used arguments of `clean()`. I have everything here except `file` and `quiet`, replace, vol, standAlone + stuff related to selecting SVC-functions

variable `var2` from `toyData`, `clean()` has identified two values that are suspected to be miscoded missing values (`999` and `NaN`), while two values were also flagged as potential outliers that should be investigated more carefully. We can then return to Part 1, the data cleaning summary, and inspect what sorts of checks were performed on this variable. hm, delete the last part? or use different variable? Only these exact two checks were performed.

Of course,

2.1. Controlling contents

By the *contents* of the cleanR output, we refer to every step that actually involves a function being called on variables from the dataset. There are three stages in which this occurs (with ref. to figure/flowchart?):

Part 1

Data cleaning summary

The dataset examined has the following dimensions:

Feature	Result
Number of rows	15
Number of variables	6

Checks performed

The following variable checks were performed, depending on the data type of each variable:

	character	factor	labelled	numeric	integer	logical
Identify miscoded missing values	×	×	×	×	×	
Identify prefixed and suffixed whitespace	×	×	×			
Identify levels with < 6 obs.	×	×				
Identify case issues	×	×				
Identify misclassified numeric or integer variables	×	×				
Identify outliers				×	×	

Please note that all numerical values in the following have been rounded to 2 decimals.

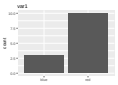
1

Part 2

Variable list

var1

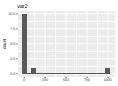
Feature	Result
Variable type	factor
Number of missing obs.	2 (13.33 %)
Number of unique values	2
Mode	"red"



- Note that the following levels have at most five observations: 'blue'.

var2

Feature	Result
Variable type	numeric
Number of missing obs.	3 (20 %)
Number of unique values	8
Median	4.5
1st and 3rd quartiles	1.75; 6
Min. and max.	1; 999



- The following suspected missing value codes enter as regular values: '999', 'NaN'.
- Note that the following possible outlier values were detected: '82', '999'.

2

Figure 1: Example output from running XXXX on the testData dataset. First a summary of the full dataset,.. Maybe fix these pdfs such that they are one file with no blankspace? I.e. basically we want the .html-version of this file printed as pdf.

1. In the precheck functions
2. In the summarize/visualize/check (SVC) step
3. In the multivariate visuzalizations Or whatever, in the stuff that we have not yet implemented. I'm pretending like this stage doesn't exist below.

Each of these stages are controllable using appropriate function parameters in `clean`. In the above, we presented the default **cleanR** settings and how to tweak them into providing a slightly different data cleaning outputs. However, if for instance the dataset at hand requires completely different visualizations, more control is needed. **cleanR** uses three different types of functions for performing all stages in the above, namely `summaryFunctions`, `visualFunctions` and `checkFunctions`. They each have distinct input-output structures and most instances of the functions are build as `S3` generics with methods for different data classes. By understanding how to construct proper `summaryFunctions`, `visualFunctions` and `checkFunctions`, the entire contents of the **cleanR** output is at your hands. Therefore, this section is dedicated to introducing each in turn.

Something like a figure that gives an overview of all summaryFunctions, ... available in standard cleanR, including default settings and some kind of description of where they are called (in precheck or SVC step). Also something about what datatypes they can each be called on. Also, more text here about the relationship between the two (three) stages and the three function types.

2.2. Something about what check, visual and summary functions are available

- Introduce the relevant `clean`-arguments and the `defaultWhateverSummaries` etc.-functions
- Introduce `allSummaryFunctions()` etc. and present a table corresponding to the output of this call
- Small example

3. Something about the interactive cleanR mode

While overview documents are great for documenting our work, they might not feel very natural for actually performing data cleaning or data wrangling based on the results we found. Therefore, `cleanR` also provides more standard `R` interactive tools, such as functions that print results to the console or saves it in a variable for later use. Or rather, the functions used by `clean()` to do actual data summaries, visualizations and checks, as described above, can also be used interactively. segway

Maybe write here that this section introduces the three functions: `summarize()`, `check()` and `visualize()`?

3.1. Blabla: An example

Let's say we wish to look further into a certain variable, namely `var2`, from `toyData`. The data cleaning summary found some issues, and we would like to recall what these issues were. This is done using the command

```
> check(toyData$var2)

$identifyMissing
The following suspected missing value codes enter as regular values: 999, NaN.
$identifyOutliers
Note that the following possible outlier values were detected: 82, 999. $
```

delete that last $, it's only there to make my TeX editor stop highlighting everything. Note: $s cannot be escaped here, as Schunk is verbatim.

Note that the arguments specifying which checks to perform, as described in XXXX above, are in fact passed to `check()`, and thus they can also be used here. For instance, if we only want the result of the check for miscoded missing values, we write

```
> check(toyData$var2, numericChecks = "identifyMissing")

$identifyMissing
The following suspected missing value codes enter as regular values: 999, NaN.
```

An equivalent way to call only a single, specific `checkFunction` is by using it directly on the variable, i.e.

```
> identifyMissing(toyData$var2)

The following suspected missing value codes enter as regular values: 999, NaN. $
```

delete that last $

The result of a `checkFunction` is an object of class `checkResult`. By using the structure function, `str()`, we can look further into its components:

```
> missVar2 <- identifyMissing(toyData$var2)
> str(missVar2)

List of 3
 $ problem      : logi TRUE
 $ message      : chr "The following suspected missing value codes enter as
                  regular values: \\\"999\\\", \\\"NaN\\\"."
 $ problemValues: num [1:2] 999 NaN
 - attr(*, "class")= chr "checkResult"
```

The most important thing to note here is that while the printed message is made for human reading, the actual values of the variable causing the issue are still attainable. If we for instance decide that the values `999` and `NaN` in `var2` are in fact miscoded missing values, we can replace them easily:

```
> toyData$var2[toyData$var2 %in% missVar2$problemValues] <- NA
> identifyMissing(toyData$var2}

No problems found. $
```

remove that last $. This is quite a practical example, but it essentially also gives the basic structure for misusing cleanR by generating a loop that does autocleaning... What do you think? Perhaps those who know how to make loops also know that they shouldn't do automatic data cleaning?

-

4. Extending cleanR

Though the discussion in the above paints a picture of cleanR as a user-friendly package which requires practically no knowledge of R, one should not be mistaken to think that it is not customizable. In fact, the main function of cleanR, `clean`, is mainly a tool for formatting the results from various checking-, summary- and visualizaion functions as described... blabla. Thus, the actual work underlying a cleanR output file can be anything or nothing - depending on the arguments given to `clean`. This section consists of three parts. We commence this section with an overview of how contents are controlled in **cleanR** in terms of what cleaning steps are performed. Secondly, we turn to the possibilities of customizing what the final cleanR file looks like. Lastly, we go through an example of how to put all these customization options together in practice.

Writing a summaryFunction

Though **cleanR** provides a special class for `summaryFunction`s, there really is nothing special about these functions: They are nothing but regular functions with a certain input/output-structure. Specifically, they all follow the template below:

```
mySummaryFunction <- function(v) {
        res <- [result of whatever summary we are doing]
        list(feature = "Feature name", result = res)
}
```

and we recommend furthermore adding them to the overview of all summary functions by converting them to proper `summaryFunction` objects:

```
mySummaryFunction <- summaryFunction(mySummaryFunction,
        description = "Some text describing what your summaryFunction does")
```

which adds the new function to the output of `allSummaryFunctions()`. Note that `v` is a vector and that `res` should be either a character string or something that will be printed as one. In other words, e.g. integers are allowed, but matrices are not. Though a lot of different things

can go into the `summaryFunction` template, we recommend only using them for summarizing the features of a variable, and leaving tests and checks for the `checkFunction`s (presented below).

Writing a visualFunction

`visualFunction`s are the functions that produce the figures of a **cleanR** output document. Writing a visual function is slightly more complicated than writing a summary function. This follows from the fact that visualFunctions need to be able to output standalone code for plots in order for `clean` to build standalone rmarkdown files. We recommend using the following structure:

```
myVisualFunction <- function(v, vnam, doEval) {
        thisCall <- call("[the name of the function used to produce the plot]",
                v, [additional arguments to the plotting function])
        if (doEval) {
                return(eval(thisCall))
        } else return(deparse(thisCall)
}

myVisualFunction <- visualFunction(myVisualFunction,
        description = "Some text describing your visualFunction")
```

In this function, `v` is the variable to be visualized, `vnam` is its name (which should generally be passed to `title` or `main` arguments in plotting functions) and `doEval` controls whether the output is a plot (if `TRUE`) or a character string of standalone code for producing a plot (if `FALSE`). The latter `doEval` setting is not strictly necessary for its use in `clean`, but it makes it easier to assess what visualization options are available. In either case, it should be noted that all the parameters listed above, `v`, `vnam` and `doEval`, are mandatory, so they should be left as is (as are?), even if you do not want to use them. As with `summaryFunction`s, an overview of all available `visualFunction`s in the environment can be obtained by calling

```
allVisualFunctions()
```

and by calling

```
allVisual(v, vnam, output = "html")
```

an overview of all plotting options applied on `v` is produced and opened as a html document for easy comparison. Should we mention the side effect of producing .rmd and .html files on disc?

Writing a checkFunction

The last, but also most important, **cleanR** function type is the `checkFunction`. These are the functions that flag issues in the data and control the flow of the overall data cleaning process in the precheck stage. A `checkFunction` can be written using the following template:

```
myCheckFunction <- function(v) {
        [do your check]
        problem <- [is there a problem? TRUE/FALSE]
        problemValues <- [vector of values in v that are problematic]
        problemStatus <- list(problem = problem, problemValues = problemValues)

        problemMessage <- "[The message that should be printed prior to listing
                        problem values in the cleanR output]"

        outMessage <- messageGenerator(list(problem = problem,
                problemValues = problemValues, message = problemMessage))

        list(problem = problem, message = message) #problem is TRUE/FALSE,
                # message is a text string
}
myCheckFunction <- checkFunction(myCheckFunction,
        description = "[A description of your checkFunction]")
```

Only the input parameter (`v`) and the output format strictly has to follow this structure. However, we recommend using `messageGenerator` for consistent styling of all `checkFunction` messages. This function simply pastes together the `problemMessage` and the `problemValues`, with the latter being quoted and sorted alphabetically. Note that printing quotes in rmarkdown requires an extensive amount of character escaping, so opting for `messageGenerator` really is the easiest solution.

While the descriptions of `summaryFunctions` and `visualFunctions` are only for internal use in the `allSummaryFunctions()` and `allVisualFunctions()` outputs, respectively, `checkFunction` descriptions are actually visible in the **cleanR** output document. These are the brief descriptions presented in Part 1 eh? Is this clear? Whatever we call this section in the above in the output document. If a `checkFunction` does not have a description (for instance, if it is just a regular function using the `checkFunction` input/output-structure), the function name will be printed instead of the description.

Eksempel

```
> characterFoo <- function(v) {
+     if (substr(substitute(v), 1, 1) == "_") {
+         out <- list(problem=TRUE, message="Note that the variable name begins with \\_")
+     } else out <- list(problem=FALSE, message="")
+     out
+ }
> class(characterFoo) <- "checkFunction"
> attr(characterFoo, "description") <- "I really hate underscores"
> #clean(testData, characterChecks=c(defaultCharacterChecks(), "characterFoo"))
>
```

5. Something like examples

5.1. Cleaning large datasets

If the dataset becomes very large, the standard use of `clean()` outlined above might not be ideal. If there is a large number of variables, production of the `rmarkdown` document might be very slow, while an extensive amount of observations generally makes rendering of this document into pdf or html very slow. In this section, we give a few practical examples of ways to deal with large data, while wishing to still produce (potentially very long) data cleaning overview documents. Note that the interactive tools of `cleanR` can be used as usual or sequentially in small subsets of the large dataset.

Dropping the figures

Though figures give a nice overview of each variable, they are also quite heavy objects in terms of memory allocation. Therefore, it might be beneficial to not include figures in the `cleanR` outputs for very large datasets. This is controlled via the `mode` argument:

```
> clean(toyData, mode = c("summarize", "check")
```

Economic memory use

Another solution, which is especially relevant to Windows users due to the unfortunate combination of memory control in this operating system and RStudio And also just R, right? There's got to be a nice reference on this..?, is simply splitting the two steps performed by `clean`, namely producing the `rmarkdown` file and rendering it afterwards. If the `rmarkdown` file is very long, as it will typically be in very large datasets, having this file opened in memory waists precious memory capacities. Therefore, we advice users to instead split the two steps. This can be done in the following manner:

```
> clean(toyData, render = FALSE}
> render("cleanR_toyData.Rmd", quiet = FALSE)
```

This also deals with the fact that `cleanR` can produce `rmarkdown` files that supersedes the upper size limit of RStudio, which is currently find number GBs (using RStudio version 1.0.44). Is this maybe too editor specific? A lot of people do use RStudio....

5.2. Use cleanR for problem flagging

If the data is large, but memory issues and computation time are less of an issue than the human time it takes to look through the data cleaning document, a viable solution might be to not include all information about all variables. Or even for more reasonably sized datasets, sometimes a brief overview of the most pressing issues can be useful. This can be achieved by using the `onlyProblematic` argument in `clean()`. By specifying `onlyProblematic = TRUE`, only variables that raise a flag in the checking steps will be summarized. But perhaps we are not even interested in obtaining general information about these variables, but only in getting a quick overview of the problems, they might have. This can be done by also controlling the `mode` argument:

```
> clean(toyData, onlyProblematic = TRUE, mode = c("check"))
```

Now only the checking results are printed, and only for variables where problems were identified. An even more minimal output can be generated by also leaving out the checking results - then `clean()` essentially just produces a list of the variable names that should be investigated further:

```
> clean(toyData, onlyProblematic = TRUE, mode = NULL)
```

Of course, this can also be done without generating an overview document, by use of the `check()` function:

```
> toyChecks <- check(toyData)
> foo <- function(x) {
>   any(sapply(x, function(y) y[["problem"]]))
> }
> sapply(toyChecks, foo)

 var1  var2  var3  var4  var5  var6
 TRUE  TRUE  TRUE  TRUE  TRUE FALSE
```

drop this last bit? too technical?

5.3. Include cleanR document in other files

6. Appendix Something

Part 1

Data cleaning summary

The dataset examined has the following dimensions:

Feature	Result
Number of rows	15
Number of variables	6

Checks performed

The following variable checks were performed, depending on the data type of each variable:

	character	factor	labelled	numeric	integer	logical
Identify miscoded missing values	×	×	×	×	×	
Identify prefixed and suffixed whitespace	×	×	×			
Identify levels with < 6 obs.	×	×				
Identify case issues	×	×				
Identify misclassified numeric or integer variables	×	×				
Identify outliers				×	×	

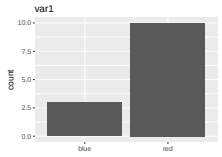
Please note that all numerical values in the following have been rounded to 2 decimals.

Part 2

Variable list

var1

Feature	Result
Variable type	factor
Number of missing obs.	2 (13.33 %)
Number of unique values	2
Mode	“red”

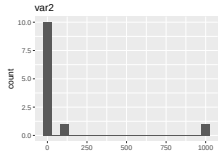


- Note that the following levels have at most five observations: "blue".

---

var2

Feature	Result
Variable type	numeric
Number of missing obs.	3 (20 %)
Number of unique values	8
Median	4.5
1st and 3rd quartiles	1.75; 6
Min. and max.	1; 999

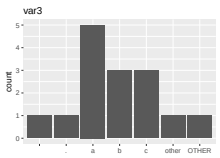


- The following suspected missing value codes enter as regular values: "999", "NaN".
- Note that the following possible outlier values were detected: "82", "999".

---

var3

Feature	Result
Variable type	factor
Number of missing obs.	0 (0 %)
Number of unique values	7
Mode	“a”

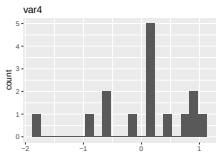


- The following suspected missing value codes enter as regular values: " ", ".".
- The following values appear with prefixed or suffixed white space: " ".
- Note that the following levels have at most five observations: " ", ".", "a", "b", "c", "other", "OTHER".
- Note that there might be case problems with the following levels: "other", "OTHER".

---

var4

Feature	Result
Variable type	numeric
Number of missing obs.	0 (0 %)
Number of unique values	15
Median	0.1
1st and 3rd quartiles	-0.38; 0.6
Min. and max.	-1.84; 1.02



- Note that the following possible outlier values were detected: "-1.84".

---

var5

- The variable is a key (distinct values for each observation).

---

var6

- The variable only takes one (non-missing) value: "1". The variable contains 0 % missing observations.

Affiliation:

Claus Thorn Ekstrøm
Biostatistics, Department of Public Health
University of Copenhagen
Denmark
E-mail: ekstrom@sund.ku.dk
URL: http://staff.pubhealth.ku.dk/~ekstrom/

Journal of Statistical Software http://www.jstatsoft.org/
published by the Foundation for Open Access Statistics http://www.foastat.org/
MMMMMM YYYY, Volume VV, Issue II *Submitted:* yyyy-mm-dd
doi:10.18637/jss.v000.i00 *Accepted:* yyyy-mm-dd

	Description	Relevant variable classes						
		C	F	I	L	B	N	D
`summaryFunctions`								
`centralValue`	Compute median or mode	×	×	×	×	×	×	×
`countMissing`	Compute ratio of missing observations	×	×	×	×	×	×	×
`minMax`	Find minimum and maximum values			×			×	×
`quartiles`	Compute 1st and 3rd quartiles			×			×	
`uniqueValue`	Count number of unique values	×	×	×	×	×	×	×
`variableType`	Data class of variable	×	×	×	×	×	×	×
`visualFunctions`								
`basicVisual`	Histograms and barplots using graphics	×	×	×	×	×	×	×
`standardVisual`	Histograms and barplots using ggplot2	×	×	×	×	×	×	×
`checkFunctions`								
`identifyCaseIssues`	Identify case issues	×	×					
`identifyLoners`	Identify levels with < 6 obs.	×	×					
`identifyMissing`	Identify miscoded missing values	×	×	×	×	×	×	
`identifyNums`	Identify misclassified numeric or integer variables	×	×					
`identifyOutliers`	Identify outliers			×		×		
`identifyOutliersTBStyle`	Identify outliers (Turkish Boxplot style)			×		×		
`identifyWhitespace`	Identify prefixed and suffixed whitespace	×	×		×			
`isCPR`	Identify Danish CPR numbers	×	×	×	×	×	×	
`isEmpty`	Check if the variable contains only a single value	×	×	×	×	×	×	
`isKey`	Check if the variable is a key	×	×	×	×	×	×	

Table 2: Blabla, mention that C is character, F is factor, I is integer, L is labelled, B is logical (boolean), N is numeric and D is Date.
Also: Check that everything in here is correct (i.e. corresponds to the output of allSummaryFunctions() etc

	`summaryFunction`	`visualFunction`	`checkFunction`
Input (required)	`v` - a variable vector `...`	`v` - a variable vector `vnam` - the variable name (as character string) `doEval` - a logical (`TRUE`/`FALSE`) controlling the output type of the function	`v` - a variable vector `nMax` - an integer (or `Inf`), controlling how many problematic values are printed, if relevant `...`
Input (optional)	`maxDecimals` - number of decimals printed in outputted numerical values	-	`maxDecimals` - number of decimals printed in outputted numerical values
Purpose	Describe some aspect of the variable, e.g. a central value, its dispersion or missingness.	Produce a distribution plot.	Check a variable for a specific issue and, if relevant, identify the values in the variable that cause the issue.
Output (required)	A list with entries `$feature` Label for the summary value (as character string) `$result` The result of the summary (as character string)	A character string with `R` code for producing a plot. This code should be standalone, i.e. should include the data if necessary.	A list with entries `$problem` - a logical identifying whether an issue was found `$message` - character string (possibly empty) decribing the issue that was found, properly escaped and ready for use in `rmarkdown`
Output (recommended)	A `summaryResult` object (i.e. an attributed list with entries `$feature`, `$result` and `$value`, the latter being the values from `$result` in their original format).	If `doEval` is `TRUE`: A plot that will be opened by the graphic device in `R`. / If `doEval` is `FALSE`: A text string with `R` code, as described above.	A `checkResult` object (an attributed list with entries `$problem`, `$message` and `$problemValues`, the latter being either `NULL` or the problem causing values, as they were found in `v`, whichever is relevant.
Tools available for producing the function	`summaryResult()`	-	`messageGenerator()` `checkResult()`

Table 3: blabla. Fix formatting! Maybe use multirow and raggedRight stuff?