



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 customizable 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 analyst 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`

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

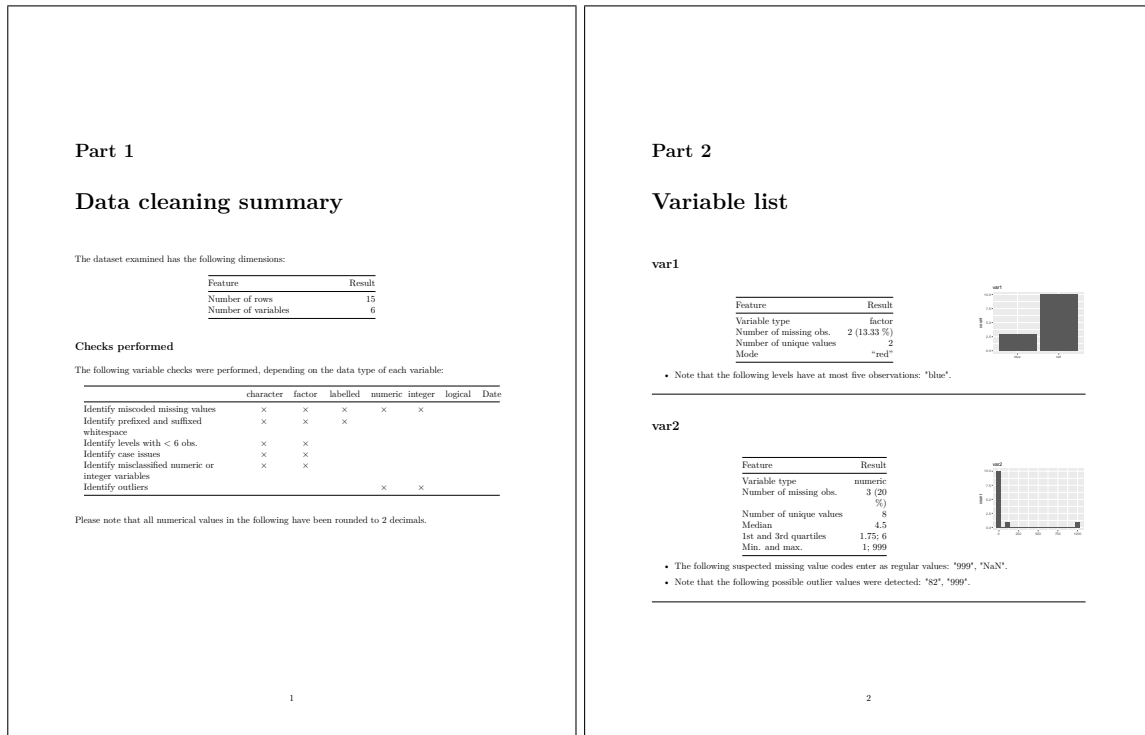


Figure 1: Example output from running `clean()` on the `toyData` dataset. First a summary of the full dataset,.. Maybe fix these pdfs such that they are one file with no blankspace? I.e. basically the .html-version of this file printed as pdf. This version is too small.

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
```

Input A dataset

- This should be of type `data.frame` or `tibble`

Create contents For each variable in the dataset, do the following:

Stage 1: Pre-checks

- Is the variable suitable for summarization, visualization and checks?
Yes: Go to stage 2
No: Move on to the next variable

Stage 2: SVC-steps

Summarize Call `summarize()` to produce a summary table describing the variable. What features enter this table depends on the data class of the variable.

Visualize Call `visualize()` to produce a plot visualizing the distribution of the variable.

Check Call `check()` to apply quality- and error checks to the variable. What checks are used depends on the data class of the variable.

Output Files for a overview document

- Always a `rmarkdown` (.Rmd) file
- Possible also a html or pdf file

Figure 2: quite odd having a figure like this? maybe do a flowchart?

	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 (in the working directory) 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 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.*

metatext here: blabla more can be done blabla...

Though the `clean()` function is very easy to use, it should not be mistaken to be inflexible. A large number of function arguments allows for the cleaning overview document to be molded as the user sees best. Below, we present a selection of these arguments in sections `xx` and `xxx`. We do this under two distinct headlines to emphasize that two levels of control are available when using `clean()`. Either, we can control the `[something]` - *essentially, this is everything but SVC-functions* - or we can control what quality assessments and descriptions each variable are subjected to. In order to understand this distinction, a brief glimpse of the inner structure of `clean()` might be useful. This can be obtained from Figure 2. *segway to next bit or just something to end the paragraph*

2.1. Controlling `[something]`

Describe arguments in the table...

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 <i>Data cleaning summary</i> ?	TRUE
Control SVC-step mode	What steps should be performed for each variable (out of the three possibilities <i>summarize</i> , <i>visualize</i> , <i>check</i>)?	c("summarize", "visualize", "check")
labelled_as	How should variables of class <code>labelled</code> be handled (as factors, is missing values or by ignoring labels)?	"factor"
smartNum	Should numerical values with only a few unique levels be flagged and 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()`. Note: I have everything here except `file` and `quiet`, `replace`, `vol`, `standAlone` + stuff related to selecting SVC-functions (dealt with in next subsection). Fix formatting for `onlyProblematic`.

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

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

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 two stages in which this occurs (with ref. to figure/flowchart?):

1. In the precheck functions
2. In the summarize/visualize/check (SVC) step

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.

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

	Description	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


```
> 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?

- Mention `allCheckFunctions()` etc. again here
- Mention `check()`, `visualize()` and `summarize()` modes for `data.frames`

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 visualization 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`. Specifically, user made functions can be added to the SVC-function arguments discussed above, e.g. `factorSummaries`, `allVisuals` or `numericChecks`: All that is needed is to specify their names, just like the names of the build-in SVC functions are specified. However, not just any function can be called from these three steps, and therefore, we will now present how `summaryFunctions`, `visualFunctions` and `checkFunctions` are made. [Mention something about the interactive mode here as well?](#)

This section consists of two parts. First, we describe how the `clean` function can be extended by adding custom prechecks, summaries, checks and visualizations. In order to do this, one needs to produce small functions that obey to the syntax of `cleanR`. This can be done with different levels of strictness. If the custom functions are only to be used by `clean()`, only the input/output structures of the functions need to be restricted. However, by adding a few extra tweaks to these functions, the full machinery of `cleanR` becomes available for the new, user-made functions, just as it is for the build-in functions. The presentation below is given in the format of pseudo-code function templates, designed for getting the full functionality, but please note that [Table 3](#) serves as a reference to the minimal requirements, while also presenting the "full" versions of the function types.

After this abstract, templatic overview of the internals of the three SVC functions is given, we turn to a worked example of how to use custom made functions in practice in [ref to worked example section](#). Here, four new SVC functions are defined and used, both interactively and in `clean()`.

4.1. Writing `cleanR` extensions: Function templates

`metatext`

Writing a summaryFunction

Though **cleanR** provides a dedicated class for **summaryFunctions**, 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]
  summaryResult(list(feature = "Feature name", result = res))
}
```

The last function called, **summaryResult()**, changes the class of the output and thereby makes a **print()** function available. 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 **checkFunctions** (presented below).

Though following the template above is sufficient for using **mySummaryFunction** in **clean()**, we recommend furthermore adding it to the overview of all summary functions by converting it to a proper **summaryFunction** object. This is done by writing

```
mySummaryFunction <- summaryFunction(mySummaryFunction,
  description = "Some text describing what your summaryFunction does",
  classes = c([the data types that this function is intended to be used for]))
```

which adds the new function to the output of **allSummaryFunctions()**. One comment should be devoted to the two attributes of a **summaryFunction**. If **description** is left unspecified, the name of the function (in this case, **mySummaryFunction**) will be filled in. What happens if the **classes** argument depends on the type of **mySummaryFunction**. If **mySummaryFunction** is a S3 generic function with associated methods, the call to **summaryFunction()** will automatically produce a vector of the names of the classes for which the function can be called. If **mySummaryFunction** is not an S3 generic and **classes** is left unspecified, the attribute will simply be empty. Note that the helper function **allClasses()** might be useful for filling out the **classes** argument, as it simply lists all available classes in **cleanR**:

```
> allClasses()

[1] "character" "Date"      "factor"    "integer"   "labelled"
[6] "logical"   "numeric"
```

write something here, don't end paragraph with code? Also, maybe move the **allClasses()** stuff somewhere else, it doesn't really belong under this header.

Writing a visualFunction

visualFunctions are the functions that produce the figures of a **cleanR** output document. Writing a **visualFunction** is slightly more complicated than writing a **summaryFunction**. 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",
  classes = c([the data types that this function is intended to be used for]))
)

```

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 `doEval = TRUE` setting is not strictly necessary for a `visualFunction`'s use in `clean`, but it makes it easier to assess what visualization options are available, and obviously, it also allows for interactive usage of `myVisualFunction`. 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, even if they are not in use.

Writing a checkFunction

The last, but perhaps most important, `cleanR` function type is the `checkFunction`. These are the functions that flag issues in the data in the check step and control the overall flow of the data cleaning process in the precheck stage. A `checkFunction` can follow two overall structures, depending on the type of check. Either, it tries to identify problematic values in the variable (as e.g. `identifyMissing()` does) or it performs a check concerning the variable as a whole (e.g. the functions used for prechecks and the function `identifyNums()`). We present templates for both types of `checkFunctions` below separately, but it should be emphasized that formally, they belong to the same class.

First, a template for the full-variable check function type:

```

myFullVarCheckFunction <- function(v, ...) {
  [do your check]
  problem <- [is there a problem? TRUE/FALSE]
  message <- "[message describing the problem, if any]"
  checkResult(list(problem = problem,
    message = message,
    problemValues = NULL))
}

myFullVarCheckFunction <- checkFunction(myFullVarCheckFunction,
  description = "Some text describing your checkFunction",
  classes = c([the data types that this function is intended to be used for]))
)

```

Again, as with `summaryFunctions` and `visualFunctions`, the change of function class by use of `checkFunction()` is not strictly necessary. Note however, that if `myFullVarCheckFunction` is to be used in the SVC step in `clean()`, the description attribute will be printed in the overview table in *Data cleaning summary*.

If problematic values are to be identified, the template from above should be expanded to follow a slightly more complicated structure:

```
myProbValCheckFunction <- function(v, nMax, maxDecimals, ...) {
  [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, ending with a colon]"

  outMessage <- messageGenerator(problemStatus, problemMessage, nMax)

  checkResult(list(problem = problem,
    message = outMessage,
    problemValues = problemValues))
}

myProbValCheckFunction <- checkFunction(myProbValCheckFunction,
  description = "[Some text describing your checkFunction]",
  classes = c([the data types that this function is intended to be used for])
)
```

In this template, the argument `maxDecimals` is not used. This argument should be used to round off the `problemValues` passed to `messageGenerator()`, if they are numerical. This is done by substituting the `problemStatus` assignment above with the following code:

```
problemStatus <- list(problem = problem,
  problemValues = round(problemValues, maxDecimals))
```

Another noteworthy component of the template is the usage of the helper function `messageGenerator()`, which aids consistent styling of all `checkFunction` messages. This function simply pastes together the `problemMessage` and the `problemValues`, with the latter being quoted and sorted alphabetically. If `nMax` is not `Inf`, only the first `nMax` problem values will be pasted onto the message, accompanied by a comment about how many problem values were left out. Note that printing quotes in `rmarkdown` requires an extensive amount of character escaping, so opting for `messageGenerator()` really is the easiest solution.

4.2. A worked example

We will now build four new functions and show both how they can be used interactively and how they can be integrated with the `clean()` function. These four new functions are:

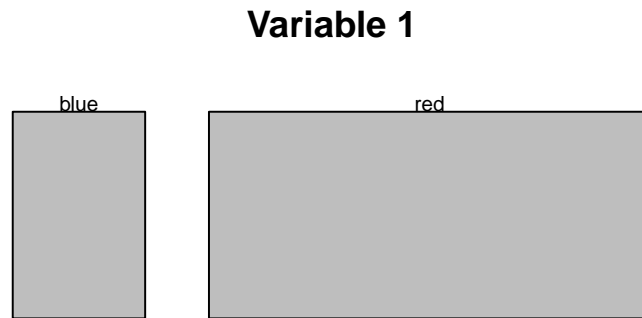


Figure 3: something, note that code for producing this plot is available in `./latex/codeForArticle.R`

isID A new `checkFunction` intended for use in the precheck-step. This function checks whether a variable consists exclusively long (> 10) entries that are all of equal length, as this might be personal identification codes that we do not wish to print out in the data summary.

mosaicVisual A new `visualFunction` that produces so-called mosaic plots. This function will be used in the *visualize* step of `clean()`.

countZeros A new `summaryFunction` that counts the number of occurrences of the value 0 in a variable. This function will be used in the *summarize* step of `clean()`.

identifyColons A new `checkFunction` that flags variables that colons that appear before and after alphanumerical characters, e.g. practical for identifying autogenerated interaction effects. This function will be used in the *check* step of `clean()`.

These functions are defined in turn below, and afterwards, an example of how they can be called from `clean()` is provided.

isID - a new checkFunction without problem values

First, let's define the `isID` function. As this function is not supposed to list problematic values in the variable, it falls within the category of `checkFunctions` represented by `myFullVarCheckFunction()` in the above. We do not particularly wish to use this function interactively, so we will stick to the minimal requirements of a `checkFunction` used in `check()` (see Table 3). The function can then be defined by

```
isID <- function(v, nMax = NULL, ...) {
  out <- list(problem = FALSE, message = "")
```

```

if (class(v) %in% setdiff(allClasses(), c("logical", "Date"))) {
  v <- as.character(v)
  lengths <- c(nchar(v))
  if (all(lengths > 10) & length(unique(lengths)) == 1) {
    out$problem <- TRUE
    out$message <- "Warning: This variable seems to contain ID codes."
  }
}
out
}

```

This is essentially all we need to do in order to include this function as a precheck-function in `clean()`, so we will leave it as is and move on to the next function, namely `mosaicVisual`.

mosiacVisual - a new visualFunction

We will define this function such that it gets the full `cleanR` functionality. This can be done using the code

```

mosaicVisual <- function(v, vnam, doEval) {
  thisCall <- call("mosaicplot", table(v), main = vnam, xlab = "")
  if (doEval) {
    return(eval(thisCall))
  } else return(deparse(thisCall))
}

```

This function can now be called directly or used in `clean()`. We will return to its usage in `clean()` below. Depending on the argument `doEval`, either a text string with code or a plot is produced. The plot resulting from the following call is found in Figure 4.2:

```
mosaicVisual(toyData$var1, "variable 1", doEval = TRUE) $
```

remove \$. Even though `mosaicVisual`, as written above, follows the style of a `visualFunction`, it is not yet truly one and therefore, it will not appear in a `allVisualFunctions()` call. In order to get this functionality, we need to change its object class. This can be done by writing

```

mosaicVisual <- visualFunction(mosaicVisual,
                               description = "Mosaic plots using graphics",
                               classes = allClasses())

```

Here, we use the function `allClasses()` to quickly obtain a vector of all the seven variable classes addressed in `cleanR`. Note that if `RmosaicVisual` has been an S3 generic function, this argument could have been left as `NULL` and then the classes for which methods are available would be added automatically.

As `mosaicVisual` is now a full blood `visualFunction`, it will also be included in the `allVisualFunctions()` output table:

```
> allVisualFunctions()
```

name	description	classes
mosaicVisual	Mosaic plots using graphics	character, Date, factor, integer, labelled, logical, numeric
basicVisual	Histograms and barplots using graphics	
standardVisual	Histograms and barplots using ggplot2	

Redo this output table when makeXFunction issues are fixed.

Now, we are done with the definition of `mosaicVisual` and we can turn to the next function in line, `countZeros`.

countZeros - a new summaryFunction

This `summaryFunction` in `spe` is defined in the following lines of code:

```
countZeros <- function(v, ...) {
  res <- length(which(v == 0))
  summaryResult(list(feature = "No. zeros", result = res, value = res))
}
```

Note that as this function computes an integer (the number of zeros), there is no difference between the entire `$result` and `$value`. If, on the other hand, the result had been a character string, extra formatting might be required in the `$result` entry (such as escaping of quotation marks), and in this scenario, the two entries would then differ. As the result is returned as a `summaryResult` object, a printing method is automatically called when `countZeros` is used interactively:

```
> countZeros(c(rep(0, 5), 1:100))
```

```
No. zeros: 5
```

As in the above, we change the class of this function in order to make it appear in `allSummaryFunctions()` calls. But now we wish to emphasize that the function is not intended to be called on all variable types, as zeros have different roles in `Dates` and in `logical` variables:

```
> countZeros <- summaryFunction(countZeros,
  description = "Count number of zeros",
  classes = c("character", "factor", "integer",
    "labelled", "numeric"))
```


more? don't end on code.

identifyColons - a new *checkFunction* with problem values

The last function mentioned above is `identifyColon()`. We define it using the helper function `messageGenerator` to obtain a properly escaped message, and we use `checkResult` to make it appear in `allCheckFunction()` calls:

```
identifyColons <- function(v, nMax = Inf, ... ) {
  v <- unique(na.omit(v))
  problemMessage <- "Note: The following values include colons:"
  problem <- FALSE
  problemValues <- NULL

  problemValues <- v[sapply(greexpr("[[:xdigit:]]:[[:xdigit:]]", v),
                           function(x) all(x != -1))]

  if (length(problemValues) > 0) {
    problem <- TRUE
  }

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

  checkResult(list(problem = problem,
                   message = outMessage,
                   problemValues = problemValues))
}

identifyColons <- checkFunction(identifyColons,
  description = "Identify non-suffixed nor -prefixed colons",
  classes = c("character", "factor", "labelled"))
```

As with the previous two functions, we also change its class. Note, however, that for `checkFunctions`, the function description will appear in the document produced by `clean()` (in the *Data cleaning summary* section), so now this is not only done for the sake of the `allCheckFunctions()` output.

Calling the new SVC function from clean()

Now, we are ready to use these new functions in a `clean()` call. **do this... what dataset should we use here? Include output in appendix, maybe..** The extended `cleanR` output document should have the following modifications, relative to the standard `cleanR` output:

- We want to add the new pre-check function, `isID`, to the already existing pre-checks.

- We wish to change the plot type for all variables to the new mosaic plot.
- We want the new summary function, `countZeros`, to be added to the summaries performed on all variable types but `Date` and `logical`.
- We want the new check function, `identifyColon`, to be added to the checks performed on `character`, `factor` and `labelled` variables.

These options are specified as follows:

```
> clean(dataSet,
  preChecks = c("isKey", "isEmpty", "isID"),
  allVisuals = "mosaicVisual",
  characterSummaries = c(defaultCharacterSummaries(), "countZeros"),
  factorSummaries = c(defaultFactorSummaries(), "countZeros"),
  labelledSummaries = c(defaultLabelledSummaries(), "countZeros"),
  numericSummaries = c(defaultNumericSummaries(), "countZeros"),
  integerSummaries = c(defaultIntegerSummaries(), "countZeros"),
  characterChecks = c(defaultCharacterChecks(), "identifyColons"),
  factorChecks = c(defaultFactorChecks(), "identifyColons"),
  labelledCheck = c(defaultLabelledChecks(), "identifyColons"))
```

The outputted document is found in Appendix [NUMTWO. Comment. Make appendix. Remember to change dataSet to something else, when we decide what data to use here.](#)

5. Something like examples

Finally, we will present a few examples of how to make `cleanR` solve certain issues related to data cleaning. First, we discuss the challenges related to cleaning large datasets, particularly in terms of memory use and computation speed. Next, we show how `cleanR` can be used for problem-flagging. Lastly, we discuss how the `cleanR` output document can be included in other `rmarkdown` documents as a mean to produce clear and concise documentation of a dataset. [I feel like there should be more topics here, but I'm all out of ideas...](#)

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.

Omitting 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"))
```

blabla, end?

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? On the other hand, 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. When called on a `data.frame`, this function produces a list (of variables) of lists (of checks) of lists (or rather, `checkResults`). Thus, the overall problem status of each variable can easily be unravelled using the list manipulation function `sapply()`:

```
> 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
```

and we find that only the final variable, `var6`, for which all observations have the value "Irrelevant", is problem-free. [drop this last bit? too technical?](#)

5.3. Include cleanR document in other files

Sometimes, a `cleanR` document might be a useful addition to a more general overview document, including also e.g. pairwise association plots, time series plots or exploratory analysis results. To this end, it is possible to produce a `cleanR` document that can readily be included in other `rmarkdown` files. This is done by using the `standAlone` argument in `clean`, which removes the preamble from the outputted `rmarkdown` file. Please note, that it is still necessary to indicate which `rmarkdown` type is being created; the pdf and html `rmarkdown` styles are unfortunately not identical.

If it is important that the embedded `cleanR` document can be rendered to either of these two file types, we recommend setting `twoCols = FALSE` and `output = html` in `clean()`, thereby essentially removing almost all output type specific formatting code from the generated `rmarkdown` file.

On the other hand, if a pdf document is to be produced, the a few extra lines need to be added to the preamble of the master `rmarkdown` document - otherwise, the two-column layout code will produce an error. The following is an example of how such a master document preamble might look like and how the `cleanR_toyData.Rmd` can be included:

```
---
output: pdf_document
documentclass: report
header-includes:
  - \renewcommand{\chaptername}{Part}
  - \newcommand{\fullline}{\noindent\makebox[\linewidth]{\rule{\textwidth}{0.4pt}}}
  - \newcommand{\bminione}{\begin{minipage}{0.75 \textwidth}}
  - \newcommand{\bminitwo}{\begin{minipage}{0.25 \textwidth}}
  - \newcommand{\emini}{\end{minipage}}
---

```\{r, child = 'cleanR_toyData.Rmd'\}
```

[Use proper formatting here.](#)

In the this example, the `cleanR_toyData.Rmd` file could have been created as follows:

```
> clean(toyData, standAlone = FALSE)
```

and the more minimal, html-style `rmarkdown` file described above can be produced using

```
> clean(toyData, standAlone = FALSE, output = "html", twoCols = FALSE)
```

don't end on code.

## 6. Conclusion/Concluding remarks/summary/?

Is this a thing in this journal? Otherwise, we might want to make some final remarks in the previous sections. Feels awkward to end with a bunch of code and some super specific examples...

### A. 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	Date
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				×	×		

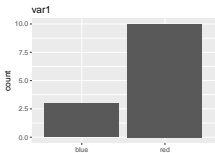
Please note that all numerical values in the following have been rounded to 2 decimals.

## Part 2

### Variable list

#### var1

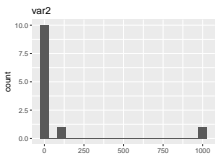
Feature	Result
Variable type	factor
Number of missing obs.	2 (13.33 %)
Number of unique values	2
Mode	“red”



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

#### var2

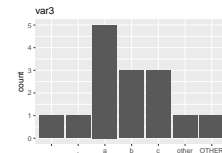
Feature	Result
Variable type	numeric
Number of missing obs.	3 (20 %)
Number of unique values	8
Median	4.5
1st and 3rd quartiles	1.75; 6
Min. and max.	1; 999



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

**var3**

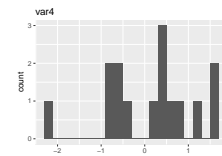
Feature	Result
Variable type	factor
Number of missing obs.	0 (0 %)
Number of unique values	7
Mode	"a"



- 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	0.33
1st and 3rd quartiles	-0.62; 0.66
Min. and max.	-2.21; 1.6



- Note that the following possible outlier values were detected: "1.12", "1.51", "1.6".

**var5**

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

**var6**

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



## B. Appendix NUMTWO

Data cleaning with user supplied extensions here

### Affiliation:

Claus Thorn Ekstrøm

Biostatistics, Department of Public Health

University of Copenhagen

Denmark

E-mail: [ekstrom@sund.ku.dk](mailto:ekstrom@sund.ku.dk)

URL: <http://staff.pubhealth.ku.dk/~ekstrom/>

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

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