# The Variable Key Data Management Framework

Paul E. Johnson<pauljohn@ku.edu>
Benjamin A. Kite<bakite@ku.edu>
Center for Research Methods and Data Analysis
University of Kansas

**January 10, 2018**

This essay describes the "variable key" approach to importing and recoding data. This method has been developed in the Center for Research Methods and Data Analysis at the University of Kansas to deal with the importation of large, complicated data sets. This approach improves teamwork, keeps better records, and reduces slippage between the intentions of principal investigators the implementation by code writers. The framework is implemented in the R (R Core Team, 2016) package `kutils`.

## 1 Introduction

The staff of the Center for Research Methods and Data Analysis has been asked to help with data importation and recoding from time to time. In one very large project, we were asked to combine, recode, and integrate variables from 21 different files. The various files used different variable names and had different, unique coding schemes. A skeptic might have thought that the firm which created the data sets intentionally obfuscated the records to prevent the comparison of variables across a series of surveys.

In projects like that, the challenge of importing and fixing the data seems overwhelming. The graduate research assistants are asked to cobble together thousands of lines of "recodes" as they rename, regroup, and otherwise harmonize the information. From a managerial point of view, that is not the main problem. We expect to spend the time of research assistants. While it may be tedious to read a codebook and write recodes, social scientists have been doing that since the 1960s. It is not all that difficult. The truly difficult part is mustering up the confidence in the resulting recoded data. How can a supervisor check thousands of recode statements for accuracy? The very extensibility of R itself–its openness to new functions and language elements–makes proof-reading more difficult. We might shift some of the proof reading duty to the principle investigators, but they sometimes are not interested in details. In the end, the responsibility for verifying the recodes falls on the project supervisors. While most supervisors with whom we are personally acquainted have nearly super-human reading skills and almost-perfect comprehension, we have documented a case in which one of them was unable to catch an error on line 827 within an R file with 2119 lines.

To reduce the risk of misunderstanding and error, we propose the *variable key procedure*. It is a systematic way to separate code writing from the process of renaming variables and

re-designating their values. The characteristics of the data are summarized in a table, a simple-looking structure that might be edited in a text editor or a spread sheet program. This simple structure, which we call the variable key, can be used by principal investigators and supervisors to designate the desired results. Once the key is created, then the data set can be imported and recoded by the application of the key's information. This does not eliminate the need to proof-read the renaming and recoding of the variables, it simply shifts that chore into a simpler, more workable setting.

This essay proceeds in 3 parts. First, the general concepts behind the variable key system are explored. Second, the four stages in the variable key procedure are outlined and illustrated with examples. Third, we offer some examples of ways to double-check the results.

## 2 Enter the Variable Key

The variable key process was first developed for a very large project for which we were hired by a commercial consulting company. As it happened, the project manager who hired us was an Excel user who did not know about R. He was given several SPSS datasets. After going through the usual R process of importing and recoding data from 6 files, the aggregate of which included more than 40,000 observations on 150 variables, we arrived at a renamed set of columns. Unfortunately, the research assistant who had done most of the work resigned in order to pursue a career as a magician.[1]

With the unavailability of our key asset, it was difficult to know for sure what was in which column. There was nobody to quickly answer questions like "which column is the respondent's sexual identity?" and "if sex is V23418, did we change 1 to male or female". The only way to find out is by hunting and pecking through a giant R file.

In order to better communicate about that project, we developed a table that looked like Table 1.

Table 1: A Small Variable Key

| name_old | name_new | values_old | values_new |
|---|---|---|---|
| V23419 | sex | 1\|2\|3 | "male"\|"female"\|"neither" |
| V32422 | education | 1\|2\|3\|4\|5 | "elem"<"hs"<"somecoll"<"ba"<"post" |
| V54532 | income | . | numeric |

It was tedious to assemble that table, but it helped quite a bit in our discussions. The vertical bars were used to indicate that the original data had discrete values. When a variable has a natural ordering, the new values were placed in order with the symbol ("<"). That table grew quite large, since it had one row per variable, but it was otherwise workable. It was popular with the client.

In the middle of preparing that summary table of recoded values, we realized that it was possible to write an R program to import the key table and use its information to recode and rename the variables. The recodes would *just happen*. If we prepared the functions properly, we had not just a table masquerading as a codebook, we had a *programmable codebook*. We wrote some functions that could import variables (as named in column 1), apply the new values (from columns 3 and 4), then apply the new name from column 2. The functions to do that are somewhat difficult to prepare, but they are very appealing from a supervisor's point of view. There will be less proof-reading to do, at least in the R code itself. Once we can validate the functions, then we never have to proof-read them again. These functions can be applied, row by row, to create a new data frame. Instead, we need to concentrate our attention on the substance of the problem, the specification of the new names and values in the table.

[1] Or graduated, we are not sure which.

In the projects where we have employed this system, we adjusted the key and the R functions to suit the particular demands of the project and the client. That was unfortunate, because we had very little accumulation of code from one project to another. However, we did accumulate experience; there were concepts and vocabulary which allowed us to understand the various challenges that might be faced. The effort to develop a standardized framework for the variable key began in 2016 with the creation of the `kutils` package for R.

The variable key process allows project supervisors to create a table that instructs the research assistants in the importation, renaming, and recoding of data. There is still a daunting problem, however, because the supervisors must create that variable key table. In a large data set, it might be arduous to simply type the old names of the variables and their observed values. In 2015 one of the graduate assistants in our lab was asked to type up a variable key and he couldn't quite believe that was a good use of his time. After some discussion, we realized that it was not necessary to type the variable key at all. We would write a function to do so. If R can import the candidate data set, then R can certainly output its column names and a roster of observed values. This lightened the workload considerably. By tabulating all of the observed variables and their values, the most tedious part of the process was done mechanically.

In the remainder of this essay, we discuss the process of creating a variable key template, revising it, and putting it to use.

## 3 Four Simple Steps

The variable key process has four steps. First, inspect an R data.frame object and create a key template file. The key template summarizes the existing state of the variables and creates "placeholders" where we might like to specify revisions. Second, edit the key template file in a spreadsheet or other program that can work with comma separate variables. Change the names, values, and designate other recodes (which we will describe next). Third, import the revised key into R. Fourth, apply the key to the data to generate a new, improved data frame. Then run some diagnostic routines.

If all goes well, we should end up with a new data frame in which

1. The columns are renamed in accordance with the instructions of the principal investigator (or supervisor).

2. The values of all variables have been recoded according to the instructions of the principal investigator (or supervisor).

Diagnostic tables are reported to clearly demonstrate the effect of each coding change, mapping out the difference between the input and the output variables.

For purposes of illustration, we have create an example data frame with various types of variables. This data frame, `mydf`, has most of the challenges that we see in actual projects. It has integer variables that need to be reorganized and turned into character or factor variables. It has character variables that might become integers or factors.

```
set.seed(234234)
N ← 200
mydf ← data.frame(
     x5 = rnorm(N),
     x4 = rpois(N, lambda = 3),
     x3 = ordered(sample(c("lo", "med", "hi"), size = N, replace=TRUE),
             levels = c("med", "lo", "hi")),
     x2 = letters[sample(c(1:4,6), 200, replace = TRUE)],
     x1 = factor(sample(c("cindy", "jan", "marcia"), 200,
             replace = TRUE)),
     x7 = ordered(letters[sample(c(1:4,6), 200, replace = TRUE)]),
     x6 = sample(c(1:5), 200, replace = TRUE),
             stringsAsFactors = FALSE)
mydf$x4[sample(1:N, 10)] ← 999
mydf$x5[sample(1:N, 10)] ← -999
```

### 3.1 Step 1. Create a Key Template

The function `keyTemplate` scans a data frame and generates a new key template. The key has 8 pieces of information about each variable. The rows of the key are named by the variables of the data frame. The 8 columns in the key are `name_old`, `name_new`, `class_old`, `class_new`, `value_old`, `value_new`, `missings`, and `recodes`. `keyTemplate` will fill `name_old`, `class_old`, and `value_old`, in the with values based on the data input, while the `new` columns will be copies of those old values. The last 2, missings and recodes, will be empty.

There are two formats for the key template, `long` and `wide` (determined by the parameter `long`). These names are drawn from terminology in R's reshape function. The long format has one row per value of each variable, while the wide format has all of the information in one row. The two key formats are intended to be interchangeable in functionality; they differ solely for convenience. Some users may prefer to edit variable information in one style. The re-importation of the key should deal gracefully with either type of variable key.

A wide format key can be produced with a call to the `keyTemplate` function like so:

```
key_wide ← keyTemplate(mydf, file = "key_wide.csv", max.levels = 5)
```

If the long argument is not specified, a wide key is the default. One can ask for a long format:

```
key_long ← keyTemplate(mydf, long = TRUE, file = "key_long.csv", max.levels = 5)
```

The key object is a data.frame.

Apart from the long argument, the `keyTemplate` function has two especially noteworthy arguments, `file` and `max.levels`. If the file argument is supplied, `keyTemplate` uses the suffix to determine storage format. Legal suffixes are `.csv`, `.xlsx`, and `.rds` (for creating comma separated variables, Excel spreadsheets, and R serialization data structures).

The `max.levels` argument is also important. This is used in the same sense that functions like `read.spss` in the foreign package use that term. There is guessing involved in deciding if we should enumerate a character or integer variable. We do want to enumerate the "Strongly Disagree" to "Strongly Agree" values of a 7 point scale, but we do not want to enumerate the first names of all study participants. If the number of discrete values exceeds `max.levels`, for which the default is 15, then the key will not enumerate them.

Table 2 demonstrates a wide key template as it is produced by `keyTemplate`. We see now why it is called a wide key; the recoding information is tightly packed into `value_old` and `value_new`. The key includes more or less obvious columns for the old and new variable names, their classes, and values of the variables. Note that the values of x5 and x4 are not enumerated because we set max.levels at 5. The `max.levels` parameter defaults to 15, so that an integer variable with less than 15 values will have each value displayed. For this example, the display of that variable key was too wide for the page, so we reduced the number of values. When the observed number of scores is above `max.levels`, the key does not try to list the individual values (compare the treatment of variables `x4` and `x6`).

```
library(kutils)
library(xtable)
```

A long key template is displayed in Table 3. The benefit of the long key is that the cells `value_old` and `value_new` are easier to navigate.

The value of `class_old` in the key is the first element in the return from the function class for a variable. There is one exception, where we have tried to differentiate integer variables from numeric variables. This is a confusing isssue in the history of R, as discussed in the R help page for the function `as.double`. ,In the *Note on names* section, that page explains an "anomaly" in the usage of term `numeric`. The R function `as.numeric` creates a double precision floating point value, not an integer. However, the is.numeric function responds TRUE for both integers and floating point values. For purposes of editing the key, it is useful to differentiate integers from floating point numbers. `kutils` includes a function named `safeInteger`. It checks the

Table 2: The Wide Key Template

| name_old | name_new | class_old | class_new | value_old | value_new | missings | recodes |
|---|---|---|---|---|---|---|---|
| x5 | x5 | numeric | numeric | . | . | | |
| x4 | x4 | integer | integer | . | . | | |
| x3 | x3 | ordered | ordered | med<lo<hi<. | med<lo<hi<. | | |
| x2 | x2 | character | character | a\|b\|c\|d\|f\|. | a\|b\|c\|d\|f\|. | | |
| x1 | x1 | factor | factor | cindy\|jan\|marcia\|. | cindy\|jan\|marcia\|. | | |
| x7 | x7 | ordered | ordered | a<b<c<d<f<. | a<b<c<d<f<. | | |
| x6 | x6 | integer | integer | 1\|2\|3\|4\|5\|. | 1\|2\|3\|4\|5\|. | | |

observed values of a variable to find out if any floating point values are present. If the aggregate deviations from integer values are miniscule, then a variable is classified as an integer. As a result, the keyTemplate function's column `class_old` should be "integer" or "numeric", and by the latter we mean a floating point number.

In some of our early projects, the variable key was in the wide format. Difficulty in editing that caused us to shift some projects to the long key. The idea that we would glide back and forth between keys created in the wide and long formats dawned on us only recently. To ease the conversion back-and-forth between the formats, we developed the functions named `wide2long` and `long2wide`. We believe these functions work effectively, but we have experienced some troubles related to the way spreadsheets store character strings. If the key in long format has a column of values "Yes","No", and "", the wide representation should be "Yes|No|", but there is some inclination to say we should have nested quotation marks, as in ""Yes"|"No"|"""". That kind of string will not generally survive importation to and export from a spread sheet at the current time.

## 3.2 Step 2. Edit the variable key

If the file argument was specified in `keyTemplate`, the work is laid out for us. One can edit a csv file in any text editor or in a spreadsheet. An xlsx file can be edited by Libre Office or Microsoft Office.

It is not necessary to change all of the values in name_new, class_new, and value_new. In fact, one might change just a few elements and the un-altered variables will remain as they were when the data is re-imported. We suggest users start small by making a few edits in the key. A principal investigator might change just a few variable names or values. In a large project, there may be quite a bit of work involved.

The `name_old` column must never be edited. Generally, `class_old` and `value_old` will not be edited either (the only exception might arise if `class_new` is either "`factor`" or "`ordered`"). The `name_new` column should include legal R variable names (do not begin name_new with a numeral or include mathematical symbols like "+" or "-"). We use R's `make.names` function to clean up errant entries, so incorrect names are not fatal.

The difficult user decisions will concern the contents of class_new and value_new. The desired variable type, of course, influences the values that are meaningful. To convert a character variable to integer, for example, it should go without saying that the value_new element should include integer values, not text strings.

The conversion of information from one type of variable into another may be more complicated than it seems. It is a bit more tricky to convert a factor into a numeric variable that uses the factor's levels as numeric values.

After experimenting with a number of cases, we believe that if `class_old` and `class_new` are elements of this the *safe class set*: `character`, `logical`, `integer`, `numeric` (same as `double`), `factor`, or `ordered`, then the re-importation and recoding of data will be more-or-less automatic. If `class_new` differs from `class_old`, and `class_new` is not an element in that 6

Table 3: The Long Key Template

| name_old | name_new | class_old | class_new | value_old | value_new | missings | recodes |
|---|---|---|---|---|---|---|---|
| x5 | x5 | numeric | numeric | . | . | | |
| x4 | x4 | integer | integer | . | . | | |
| x3 | x3 | ordered | ordered | med | med | | |
| x3 | x3 | ordered | ordered | lo | lo | | |
| x3 | x3 | ordered | ordered | hi | hi | | |
| x3 | x3 | ordered | ordered | . | . | | |
| x2 | x2 | character | character | a | a | | |
| x2 | x2 | character | character | b | b | | |
| x2 | x2 | character | character | c | c | | |
| x2 | x2 | character | character | d | d | | |
| x2 | x2 | character | character | f | f | | |
| x2 | x2 | character | character | . | . | | |
| x1 | x1 | factor | factor | cindy | cindy | | |
| x1 | x1 | factor | factor | jan | jan | | |
| x1 | x1 | factor | factor | marcia | marcia | | |
| x1 | x1 | factor | factor | . | . | | |
| x7 | x7 | ordered | ordered | a | a | | |
| x7 | x7 | ordered | ordered | b | b | | |
| x7 | x7 | ordered | ordered | c | c | | |
| x7 | x7 | ordered | ordered | d | d | | |
| x7 | x7 | ordered | ordered | f | f | | |
| x7 | x7 | ordered | ordered | . | . | | |
| x6 | x6 | integer | integer | 1 | 1 | | |
| x6 | x6 | integer | integer | 2 | 2 | | |
| x6 | x6 | integer | integer | 3 | 3 | | |
| x6 | x6 | integer | integer | 4 | 4 | | |
| x6 | x6 | integer | integer | 5 | 5 | | |
| x6 | x6 | integer | integer | . | . | | |

element set, then the user must supply a recode function that creates a variable of the requested class. Most commonly, we expect that will be used to incorporate `Date` variables.

The enumerated values in the `value_new` column should be specified in the more or less obvious way. If `class_new` is equal to `character`, `factor`, or `ordered`, then the new values can be arbitrary strings.

The `missings` and `recodes` columns are empty in the key template. The user will need to fill in those values if they are to be used. When the key is later put to use, the order of processing will be as follows. First, the values declared as missings will be applied (convert observed value to R's NA symbol). Second, if there is a recode function in the key, it is applied to a variable. Third, if there was no recode function supplied, then the conversion of discrete values by recalculation from `value_old` into `value_new` will be applied. Note that the discrete values are applied only if the recode cell is empty in the key.

The decision of whether to approach a given variable via value enumeration or a recode function is, to an extent, simply a matter of taste. Some chores that might be handled in either way. If a variable includes floating point numbers (temperatures, dollar values, etc), then we would not rely on new assignments in `value_old` and `value_new`. Truly numeric variables of that sort almost certainly call for assignment of missings and recodes by the last two cells in the variable key. However, if a column includes integers or characters (1 = male, 2 = female), one might use the enumerated values (`value_old` and `value_new`) or one could design a recode function to produce the same result. It is important to remember that if a recode function is applied, the enumerated value recoding is not. If one decides to use a recode statement, then the elements in value_old and value_new are ignored entirely, they could be manually deleted to simplify the key. (That is to say, the `max.levels` parameter is just a way of guessing how many unique levels is "too many" for an enumeration. Users are free to delete values if recodes are used.)

Despite the possibility that a factor (or ordered) variable may have many values, we believe that all of the levels of those variables should be usually be included in the key. If a variable is declared as a factor, it means the researcher has assigned meaning to the various observed values and we are reluctant to ignore them.

There is a more important reason to enumerate all of the legal values for factor variables. If a value is omitted from the key, that value will be omitted from the dataset.

Among our users, we find opinion is roughly balanced between the long and the wide key formats. One might simply try both. If the number of observed values is more than 5 or 10, editing the key in a program like Microsoft Excel is less error prone in the long key. This is simply a matter of taste, however. The disadvantage of the long format is that it is somewhat verbose, with repeated values in the name and class values. If an editor makes an error in the assignment of a block, then hard to find errors may result.

Because editing the key can be a rather involved process, we will wait to discuss the details until section 4.

### 3.3 Step 3. keyImport

Once any desired changes are entered in the variable key, the file needs to be imported back into R. For that purpose, we supply the `keyImport` function. As in `keyTemplate`, the file argument's suffix is used to discern whether the input should be read as .csv, .xlsx, or .rds. It is not necessary to specify that the key being imported is in the long or wide format. `keyImport` includes heuristics that have classified user-edited keys very accurately.

The returned value is an R data frame that should be very similar to the template, except that the new values of `name_new`, `class_new`, and `value_new` will be visible.

In order to test this function with the `kutils` package, we include some variable keys. The usage of those keys is demonstrated in the help page for `keyImport`. In addition to the mydf

toy data frame created above, we also include a subset of the US National Longitudinal Survey in a data frame named `natlongsurv`.

### 3.4 Step 4. Apply the imported key to the data

The final step is to apply the key to the data frame (or some other data frame that may have arrived in the interim). The syntax is simple

```
mydf.cleaned ← keyApply(mydf, mydf.keylist)
```

Because the default value of the argument `diagnostic` is TRUE, the output from `keyApply` is somewhat verbose. After we have more feedback from test users, we will be able to quiet some of that output.

The diagnostic output will include information about mismatch between the key and the data itself. If variables that are included in the key that are not included in the new data set, there will not be an error, but a gentle warning will appear. Similarly, if the observed values of an enumerated variable are not included in the variable key, there will be a warning.

The diagnostic will also create a cross tabulation of each new variable against its older counterpart. This works very well with discrete variables with 10 or so values, but for variables with more values it is rather unmanageable.

## 4 Editing the variable key

The work of revising the variable key can be driven by the separation of variables into two type. The variables with enumerated values–the ones for which we intend to rename or re-assign values one-by-one–are treated in a very different way than the other ones. The enumerated value strategy works well with variables for which we simply need to rename categories (e.g, "cons" to "Conservative"). Variables for which we do not do so (e.g., convert Fahrenheit to Celsius) are treated differently.

As we will see in this section, the revision of variables of the enumerated value type emphasizes the revision of the value_old and value_new columns in the key. On the other hand, the other types will depend on writing correctly formatted statements in the recode column of the variable key.

### 4.1 Enumerated variables

All of the values observed for `logical`, `factor`, and `ordered` variables will appear in the key template. Do not delete them unless the exclusion of those values from the analysis intended. For character and integer variables with fewer than `max.levels` discrete values, the observed scores will be included in `value_old`. If one wishes to convert a variable from being treated as an enumerated to a numeric type, then one can delete all values from `value_old` and `value_new`.

The recoding of discrete variables is a fairly obvious chore. For each old value, a new value must be specified. We first consider the case of a variable that enters as a character variable but we might like to recode it and also create factor and integer variants of it. In the `mydf` variable key (Table 2), we have variable `x2` which is coded `a` through `f`. We demonstrate ways to spawn new character, factor, or integer variable in Table 4. As long as name_old is preserved, as many lines as desired can be used to create variables of different types. Here we show the middle section of the revised key in which we have spawned 3 new variants of x2, each with its own name.

Table 4: Change Class, Example 1

| name_old | name_new | class_old | class_new | value_old | value_new |
|---|---|---|---|---|---|
| x2 | x2.char | character | character | a\|b\|c\|d\|f | Excellent\|Proficient\|Good\|Fair\|Poor |
| x2 | x2.fac | character | factor | a\|b\|c\|d\|f | Excellent\|Proficient\|Good\|Fair\|Poor |
| x2 | x2.gpa | character | integer | a\|b\|c\|d\|f | 4\|3\|2\|1\|0 |

In line one of Table 4, the class `character` remains the same. That line will produce a new character variable with embellished values. Line two demonstrates how to create an R factor variable, `x2.fac`, and line three converts the character to an integer variable. Remember that it is important to match the value of `class_new` with the content proposed for `value_new`. Do not include character values in a variable for which the new class will be numeric or integer.

Similarly, it is obvious to see how an integer input can be converted into either an integer, character, or factor variable by employing any of the rows seen in Table 5.

Table 5: Change Class Example 2

| name_old | name_new | class_old | class_new | value_old | value_new |
|---|---|---|---|---|---|
| x6 | x6.i100 | integer | integer | 1\|2\|3\|4\|5 | 100\|200\|300\|400\|500 |
| x6 | x6.c | integer | character | 1\|2\|3\|4\|5 | Austin\|Denver\|Nashville\|Provo\|Miami |
| x6 | x6.f | integer | factor | 1\|2\|3\|4\|5 | F\|D\|C\|B\|A |

If a variable's `class_old` is ordered, and we simply want to relabel the existing levels, the work is also easy (see Table 6). The second row in Table 6 shows that factor levels can be "combined" by assigning the same character string to several "<" separated values.

Table 6: Change Class, Example 3

| name_old | name_new | class_old | class_new | value_old | value_new |
|---|---|---|---|---|---|
| x7 | x7.grades | ordered | ordered | f<d<c<b<a | F<D<C<B<A |
| x7 | x7.passfail | ordered | ordered | f<d<c<b<a | Fail<Fail<Pass<Pass<Pass |
| x7 | x7.gpa | ordered | integer | f<d<c<b<a | 0\|1\|2\|3\|4 |

Working with ordered variables, whether as input or output, becomes more complicated if the existing data are not ordered in the way we want. In the `mydf` example, the variable `mydf$x3` was coded as an ordered variable with levels ("med", "lo", "high"). That might have been one person's idea of a joke, so we need to rearrange these as ("lo", "med", "high"). If the original ordering of the values is not consistent with the desired ordering of the new ordered factor, then we need notation that allows researchers to achieve two purposes. First, the values must be re-leveled. Second, allow for the possibility that the new values must be relabeled as well. We'd rather not proliferate new columns in the variable key or create some confusing new notation.

Reordering variable levels requires us to do something that seems dangerous. We need to edit the `value_old` to correct the ordering of the levels *as they are currently labeled*. This is the only time where we suggest that users edit the `value_old` column. In `value_new`, supply new labels in the in the correct order to parallel the newly edited `value_old` column (see Table 7).

Table 7: Reorder Values, Example 1

| name_old | name_new | class_old | class_new | value_old | value_new |
|---|---|---|---|---|---|
| x3 | x3.lo2hi | ordered | ordered | low<med<hi | low<medium<high |
| x3 | x3.passfail | ordered | ordered | low<med<hi | low<pass<pass |

The key importer will check that all "duplicated levels" are adjacent with one another, so that the values above low are grouped together.

In the long key format, the equivalent information would be conveyed by altering the ordering of the rows. For example, it is necessary to re-order the rows to indicate the lo is lower than med, and then for the new values we put in the desired names (see Table 8).

Table 8: Reorder Values, Example 2

| name_old | name_new | class_old | class_new | value_old | value_new |
|---|---|---|---|---|---|
| x3 | x3lo2hi | ordered | ordered | lo | low |
| x3 | x3lo2hi | ordered | ordered | med | medium |
| x3 | x3lo2hi | ordered | ordered | hi | high |
| x3 | x3.passfail | ordered | ordered | low | low |
| x3 | x3.passfail | ordered | ordered | medium | pass |
| x3 | x3.passfail | ordered | ordered | high | pass |

What to do about missing values. Even within our small group, there is some disagreement about this. The SAS tradition would have us enter a period, ".", in value_new for a level that is to be treated as missing. Others are tempted to use special purpose character strings like "NA" or "N/A". As long as neither "NA" nor "N/A" are legal values, that seems safe. There is another school of thought to argue that if a user wants nothing, the clearest, safest thing to do in the key is to enter nothing at all.

As a result of this diversity, the keyImport function includes an argument named na.strings (again, an argument drawn from R core funcitons). The default setting has an "all of the above" flavor, treating any of the values that any of us think might be missing as NA when the key is later put to use. If one is fastidious in editing the key and represents all value_new that should be missing as a period (hooray for SAS!), then the value of na.strings can be tightened up in the obvious way.

It is also possible to delete a level from value_old and value_new in order to indicate that it should be treated as missing, but we do not recommend this approach. This destroys an element of "book keeping" because future users of the data, and the key, might like to know that some levels were obliterated.

The missing value cell in the key can be used with these enumerated variables. Usually, writing the values to be omitted is not as easy as putting the desired missing symbol in the value_new column, but either method should work.

In a long format key, there will be many repeated rows for each variable. The entries in the `missings` and `recodes` columns are harvested from the cells corresponding to each combination of name_old and name_new, it is not necessary to retype the `missings` and `recodes` entries for each element.

The work of assigning missing values is carried out by a `kutils` function named `assignMissing`. The help page for that function has verbose commentary to explain the format of the cells that might work well in the missings column.

## 4.2 About non-enumerated variables

In the process that creates the variable key template, we think differently about the enumerated variables–ones that have meaningfully discrete values that can be reassigned one-by-one–and numeric variables. Numeric variables, whether we literally mean integers like age or the number of pennies in a jar, or floating point numbers (weight, temperature, volume, etc) are different. If we guessed right with `max.levels` when the key was created, those non-discrete variables are

included as just one row, whether the key is wide or long. In those variables, the main elements of interest for these variables are the columns labeled `missings` and `recodes`.

The instructions for specifying missing values are detailed in the help page of the `assignMissing` function in `kutils`. If the class_old is `numeric` or `integer`, there are only three types of statements allowed in the `missings` column. Legal values must must begin with the characters "<", ">", or "c". These are illustrated in Table 9. The symbols "<=" and ">=" are accepted in the obvious way.

Table 9: Missings Examples

| missings | interpretation: NA will be assigned to | example |
|---|---|---|
| > t | values greater than t | > 99 |
| >= t | values greater than or equal to t | >=99 |
| <t | values less than t | <0 |
| <=t | values less than or equal to t | <=0 |
| c(a,b) | values equal to or greater than a and less than or equal to b | c(-999,-1) |

The key specification for `recodes` is discussed in the help page of the `assignRecode` function in `kutils`. The `recodes` column takes R code and applies it to the desired variable. For example, if one wanted to transform a variable by taking its square root, this could be done by providing "sqrt(x)" in the `recodes` column. Here "x" is simply a placeholder where the name of the variable indicated in the `name_new` column will be inserted when the variable key is applied.

Table 10: Recoding Integer and Numeric Variables

| name_old | name_new | class_old | class_new | value_old | value_new | missings | recodes |
|---|---|---|---|---|---|---|---|
| x5 | x5 | numeric | numeric | | | <0 | log(x) |
| x4 | x4 | numeric | numeric | | | c(-999) | abs(x) |
| x6 | x6 | integer | integer | 1\|2\|3\|4\|5 | 1\|2\|3\|4\|5 | c(-9) | |

## 4.3 Class conversions

The problem of recoding a variable, but leaving its class the same, is mostly solved.

The conversion of variables from one class to another requires special care. R's built in functions for coercion of variables from one type to another succeed meaningfully in many cases, but not all. The coercion of an integer to a floating point number produces understandable effects. The coercion of a character to an integer is not always understandable, and the conversion of factors to integers or numeric is, well, almost always a source of concern.

We are intending to support, at minimum, the 6 safe classes, which can be used in any combination of `value_old` and `value_new`. We need to be sure that conversion from each one into the other types is handled accurately.

In the `keyApply` function, we have implemented special purpose code to handle a class change from factor to numeric, for example, in a way that preserves the levels as the new values. This is needed when the input data has values like 10, 12, 14, and "unknown". If an R function like read.table is used to import a column with those values, the default behavior will convert that to a factor variable with values "10", "12, "14" and "unknown". What the user probably wants–and what we do now–is open the possibility that "unknown" should be treated as NA and the values that appear to be numbers (but are actually text strings) are converted into integer format.

## 4.4 Unanticipated `class_new` values

By far, the most troublesome variable type is date and time information. This is difficult for a number of reasons. In order from less troublesome to more troublesome:

1. No two people seem to represent dates with the same style. Virtuous people (the authors) write 2016-12-01, while others write ambiguous strings like 12/01/2016, 01/12/2016 or 01-12-2016.

2. Computer programs absorb dates and convert them to internal numeric schemes. However, programs differ in the way they record and export date information. Most software converts dates as integers, for example. The value is the number of days since time began.

   Unfortunately, it appears that the day on which time began differs among programs. Microsoft Excel records dates as integers, and (shockingly!) the origin date differs between Excel files created in Windows and Macintosh. Mac designers were perplexed by the problem of leap years, which caused them to use the origin 1904-01-01, while in MS Windows Excel dates begin at 1900-01-01. The program SPSS uses integers for dates as well, but the origin for SPSS is 1582-10-14 (we guess most readers already knew that one, we apologize for being surprised).

   On the other hand, dates on Unix/Linux systems tend to follow the POSIX standard, so date information is interpreted as the number of days since 1970-01-01. That is a local favorite because it reminds us of Nixon and Vietnam.

3. Dates are, to a certain extent, arbitrary without time zone information. Its Tuesday, December 8 in Lawrence, Kansas, but it is Wednesday, December 9 in Moscow, Russia. If time zones are not specified, than all date information is, well, approximate. Furthermore, when scholars try to put time zone information onto existing data that was collected without it, they sometimes accidentally create impossible dates and times (February 29, 1999, for example). When R functions like `as.Date` encounter impossible dates, they return NA.

4. Even when programs seem to have absorbed dates and saved the information, even in casual testing we have found that the saved information is not retrieved in a consistent way. Some spreadsheet programs, on some computers, return values dates as integers, while the same program with a different user returns a character string.

All of this complaining is just our way of saying, "we left the recode box open, why not use it to recode your date information?" If the variable key template says that a column is "integer," but you expected a date, then all is likely to be fine if we set the origin. One needs to find out what the origin should be and tell R about it. The right recode statement is likely to be `as.Date(x, origin = "1970-01-01")`. On the other hand, if the date information is in text format, with values like "2016-11-19", then it is necessary to learn the shorthand notation for date symbols. The right recode will be something like `as.Date(x, format = "%Y-%m-%d")`.

If one sets `class_new` = `Date`, the `keyApply` function will succeed if the return value from the recode function is an R Date variable. Otherwise, an error will be raised.

## 4.5 Where more work should be done

For the most part, our work now lies in development of better diagnostics and error-checking routines.

First, we need better filtering of user-created key entries. User typographical errors occur in key preparation and more comprehensive filtering should be done. At the current time, the `keyImport` function does not include many subroutines for the validation of the key. We'd like

to filter the values requested in `class_new`, for example, but it is difficult to see how that can be done. We want to leave open the possibility that users may specify values of `class_new` that are unfamiliar to us. Another problem may arise if the number of elements in `value_old` and `value_new` are not aligned one-to-one. There will be an error when the key is put to use by `keyApply`, but we should have a stop message in `keyImport`.

Second, we need to better understand the problems that result when users enter variable keys in programs like MS Excel or Libre Office. Between programs, or versions of programs, or programs on various platforms, we notice disappointing inconsistencies. If keys are kept in "flat text" csv files, there is less danger, but there can still be some trouble because spread sheets change the way they export to csv from time to time. During the testing process, we have found that version updates Excel cause unexpected changes in the way character vectors and dates are stored. Until now, we have relied on the very popular package `openxlsx` (Walker, 2015). This does well most of the time, but we still find some inexplicable variations in imported Excel files. Sometimes, empty cells in the variable key appear as imported empty strings "", while sometimes they are imported as R NA symbols. While it is probably safe to treat an entirely empty value as a missing value in most projects, we'd rather not do so. To deal with this problem, we need to develop a standard framework for testing the quality of imported variable keys.

Third, we need more comprehensive error-checking for the accuracy of the imported data and compliance with the variable key. The variable key system works well with the data sets for which it was created, but we need more formal criteria for verifying that claim.

## 5 Discussion

When a project has a small budget, we invite the principal investigator to economize on the expenses by filling out the variable key's `name_new`, `class_new`, and `value_new` columns. There are several benefits in inviting the clients (or PIs) to be directly involved in filling in the variable key. Most importantly, they are allowed to name the variables in any way that is meaningful to them. When statistical results are obtained, it is never necessary for them to ask, "what did you mean by this variable `occupation`?" There are other benefits, however. By making the principal investigator aware of the values that are actually observed, and by offering the opportunity to specify how they ought to be recoded, a substantial element of administrative slippage is ameliorated. The variable key will specify exactly how categories are to be re-mapped, there is much less danger of an accident buried in thousands of lines of recodes.

It often happens that the raw data to be imported are provided by one of the national data centers. The variables are given exciting, meaningful column names like V34342a. It appears to be almost certain that research assistants will conclude that these names are not meaningful, so they create names that are more meaningful to them, such as *gender*, *sex*, *male*, *female*, or whatnot. The research assistants disappear into a haze of code and come out talking about the effect of income, gender, and education on educational achievement, and the principal investigator has to say, "which of those variables is income, again?" and "what's the coding on education?" A very exciting conversation then follows as one of the research assistant realizes that V34342b is the one that should have been used for gender, while V34342a indicates if the respondent ever visited Eastern Europe.

The variable key is intended to create neutral territory between principal investigators and project supervisors on one side and the research assistants on the other. The numbers are more likely to come out correctly if the names, values, and classes can be specified in one document that is accessible in numerous formats on which many eyes can be laid. The distance between a re-design of the imported values is minimized and the production cycle of the projects is accelerated. As we are frequently reminded, there are many some bright people who don't know R, it appears they are all fluent in spread sheet.

## References

R Core Team (2016). *R: A Language and Environment for Statistical Computing*. Vienna, Austria: R Foundation for Statistical Computing. ISBN 3-900051-07-0.

Walker, A. (2015). *openxlsx: Read, Write and Edit XLSX Files*. R package version 3.0.0.