read.xlx

by Antonio Fasano

Description

There is a new function in the `read.*` family, `read.xlx()`, which can read Excel xlsx workbook sheets into R data frames. Some features are:

- It can import all, one, or a selection of sheets, where specific sheets are requested by means of their name.
- Instead of importing all the sheets' cells, it can import only those comprised in a named range.
- It can distinguish between cells formatted as numbers, percent, text and dates,
- Date cells are recognised whatever the language locale.
- Blank (visual) lines are detected and automatically removed from the data frame, unless you want to keep them.
- The filter is not based on any external engine and does not requires Excel to be installed at all. It's pure R code, so you can read xlsx files on Linux systems.

Synopsis

```
read.xlx(
    file, sheets=NULL, header.sheets=FALSE, header.ranges=FALSE, ranges=NULL,
    skip=0, skipafter=FALSE, keepblanks=FALSE,
    general='numeric', morechar=FALSE, na.string="N.A.",
    time2str=FALSE, simplify=TRUE, info=FALSE)
```

file path to xlsx file.
sheets character vector with sheet names to read or NULL to read all.
header.sheets TRUE if, for all sheets, the first row is a header line to be used for column names. It can also be a logical vector whose values are TRUE (FALSE) for each sheet with (without) header.
header.ranges TRUE if, for all named ranges in `range`, the first row is a header line to be used for column names. It can also be a logical vector whose values are TRUE (FALSE) for each named range with (without) header.
ranges character vector with sheet names to read or NULL to not read use them.
skip the number of rows to skip for every sheets (and currently ranges). If there are no headers, skip the first `skip` rows; if header are set TRUE, skipped rows depend on the value of `skipafter`. It counts blank lines if `realskip==TRUE`
skipafter If TRUE skipped rows are counted after the (first used as) header row, else the rows are skipped above the headers, which will be the `skip+1`. Ignored without headers.
realskip If TRUE (default) `skip` counts blank lines, else only non-blank.
keepblanks If TRUE do not import rows or columns having only empty cells.
general How Excel General format and not explicitly managed formats are to be mapped in R. It can be to `character` or `numeric`.
morechar If FALSE, do not use directly managed cell number formats, but use always R character class.
na.string character vector of strings to be interpreted as NA values during numeric conversions.
time2str If TRUE, convert Excel Time serial to "HH:MM:SS.dec" string
simplify The output is simplified removing the enclosing list, when it would contain a single item list.
info 3 element list: `wbsheets`, vector of sheets' names; `rgsheets`, named vector whose names are the names of the workbook ranges and whose values are the names of theirsheets; `rgrefs`, named vector whose names are the names of the workbook ranges and whose values are their references.

Details

If `header.sheets` is a logical vector, its length should match the length of the workbook sheets, including empty sheets, or the length of `sheets` if this argument is not NULL. Similarly, if `header.ranges` is a logical vector, its length should match the length of `range`.

`info` is used in conjunction with `file` only.

Value Returned

If `simplify==FALSE`, a list of data frames for each sheet imported.
If `simplify==TRUE`, the list collapses to a single data frame when a single sheet is imported.

Setup

Currently the script is not in package form so, after downloading it, just source the R source with:

`source('path\to\xlx.r')`

You are done.

Use It in the Simplest Form

A sample spreadsheet should come together with this manual, `survey.xlsx` which you can use to run the same code as here.
The sample spreadsheet consists of an empty sheet and two sheets `Survey1` and `Survey2` with plain formatted data like the following:

To import the file with all sheets simply run:

```
surv=read.xlx('survey.xlsx', header.sheets=FALSE)
```

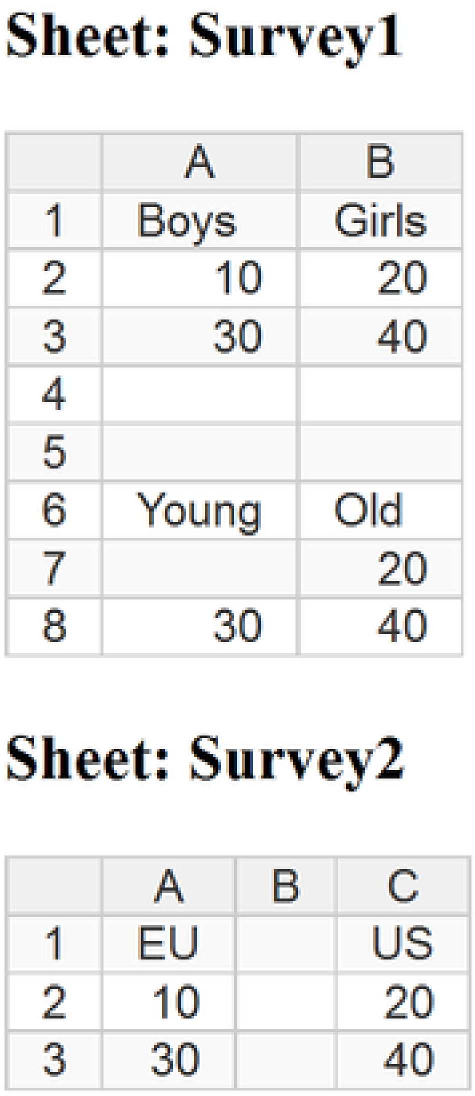
```
## ... Loading cells in sheet 1
## Cell:  11   ... Reshaping sheet 1
## ... Loading cells in sheet 2
## Cell:  6   ... Reshaping sheet 2
## ... Loading cells in sheet 3
## Aggregating sheets
## Formatting sheet(s) as data frames
## Identifying and applying prevailing styles
```

Above you see some info about the workbook being processed that we will not show anymore in the following.

The result of your import is:

```
surv
```

```
## $Survey1
##       1     2
## 1  Boys Girls
## 2    10    20
## 3    30    40
## 6 Young   Old
## 7  <NA>    20
## 8    30    40
##
```



Sheet: Survey1

	A	B
1	Boys	Girls
2	10	20
3	30	40
4		
5		
6	Young	Old
7		20
8	30	40

Sheet: Survey2

	A	B	C
1	EU		US
2	10		20
3	30		40

Figure 1: figure

```
## $Survey2
##    1  3
## 1 EU US
## 2 10 20
## 3 30 40
```

Following the general convention for the `read.*` family of functions, the sheets are converted into data frames, plus data frames are wrapped into a list. The data frame comprising the list are named like the equivalent Excel sheet, here `Survey1` and `Survey2`. Comaring the input spreadheet with the above output, you might notice that:

1. Blank sheets are, by default, not imported.
2. Because we have set the option `header.sheets=FALSE`, Excel column letters are converted to digits and set as the names of the related data frame columns.
3. Blank lines, that is whole blank rows or columns, are removed; instead single blank cells inside data tables are reported as NA.
4. When blank rows or columns are skipped, line numbers are not updated sequentially. This is done by purpose to identify the skips.

Let us analyse the output class:

```
class(surv)
```

```
## [1] "list"
```

```
names(surv)
```

```
## [1] "Survey1" "Survey2"
```

It is a list and its elements are named after the sheet names. So we check the class of its elements:

```
lapply(surv, class)
```

```
## $Survey1
## [1] "data.frame"
##
## $Survey2
## [1] "data.frame"
```

Note that when the output of `read.xlx()` consists of a single element and `simplify==TRUE`, the class of output value is simplfied to a data frame (and not a list with a single data frame).

What are the names of the (data frame) element "Survey2" of the list `surv`:

```
names(surv$Survey2) # data frame column names
```

```
## [1] "1" "3"
```

```
rownames(surv$Survey2)
```

```
## [1] "1" "2" "3"
```

As regards column names, the digits result from the above set `header.sheets=FALSE`. It is otherwise possible to take the column names from the sheet header row.

`read.xlx()` will try to match Excel formatting with suitable R classes. If formatting inside columns is inconsistent `read.xlx()` will coerce incoherent cells to the column prevailing format and issue a warning.

Import individual workbook items

We can customise the default import behaviour. For example, let us assume we want to import only the sheet "Survey2"

```
(surv= read.xlx('survey.xlsx', sheets=c("survey2"), simplify=TRUE))
```

```
##   EU US
## 2 10 20
## 3 30 40
```

As usual for a single item `sheets=c("survey2")` can be shortened as `sheets="survey2"`

Note that, respecting Excel convention, *sheet names are not case sensitive*, so `survey2` works even if the actual sheet name is `Survey2`. Anyway, bear in mind, the case used with the `sheets` argument becomes the one stored in the output, in case you later need to address it.

Another thing is that, since we asked for a single sheet, there is no need to wrap it in a worthless list, this is accomplished by the above `simplify=TRUE`. In fact now:

```
class(surv)
```

```
## [1] "data.frame"
```

Managing headers

What about using the first sheet row as the data table names?

```
surv=read.xlx('survey.xlsx', sheets=c("survey1", "survey2"), header.sheets=c(FALSE, TRUE))
```

The vector `header.sheets` identifies the imported sheets for which we want the first row to be used as a header, that is as the resulting data frame's name vector. In our case this is resp. `FALSE`, `TRUE` for "survey1", "survey2". Therefore:

```
surv
```

```
## $survey1
##       1     2
## 1  Boys Girls
## 2    10    20
## 3    30    40
## 6 Young   Old
## 7  <NA>    20
## 8    30    40
##
## $survey2
##   EU US
## 2 10 20
## 3 30 40
```

```
names(surv$survey1)
```

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

```
names(surv$survey2)
```

```
## [1] "EU" "US"
```

As you can see above, the data frame names relative to "survey1" shows the usual digits, while the names for "survey2" are taken from the first sheet row and are `"EU" "US"`.

In this way we kill two birds with one stone. First, we save time (and possibly errors) in renaming the output data frame. Secondly, since headers are string labels, using them as the first row of an otherwise numeric data frame will prevent a proper Excel-to-R class conversion.

To understand this, let us first import "survey2" without header:

```
(surv=read.xlx('survey.xlsx', sheets= "survey2", header.sheets=FALSE, simplify=TRUE))
```

```
##    1  3
## 1 EU US
## 2 10 20
## 3 30 40
```

```
class(surv[,1])
```

```
## [1] "character"
```

```
class(surv[,2])
```

```
## [1] "character"
```

As we see above, to match the class of the first rows, `read.xlx()` has to convert everything into character classes. But we need numbers:

```
(surv=read.xlx('survey.xlsx', sheets= "survey2", header.sheets=TRUE, simplify=TRUE))
```

```
##   EU US
## 2 10 20
## 3 30 40
```

```
class(surv[,1])
```

```
## [1] "numeric"
```

```
class(surv[,2])
```

```
## [1] "numeric"
```

By moving labels in the first row to data frame names, `read.xlx()` can keep numbers as such.

`header.sheets` values are recycled. So `sheets=c("survey1", "survey2"), header.sheets=TRUE` means that for both "survey1" and "survey2" the first row will be used for labeling.

It is possible to query for named ranges too. To this end, our `survey.xlsx` contains the named ranges `education` and `students`. To import them we use the `ranges` argument:

```
read.xlx('survey.xlsx', ranges=c('education', 'students'))
```

```
## $education
##       1   2
## 6 Young Old
## 7  <NA>  20
## 8    30  40
##
## $students
##      1     2
## 1 Boys Girls
## 2   10    20
## 3   30    40
```

Likewise sheets, we can now use the first rows of ranges as headers:

```
read.xlx('survey.xlsx', ranges=c('education', 'students'), header.ranges=TRUE)
```

```
## $education
##   Young Old
## 7    NA  20
## 8    30  40
##
## $students
##   Boys Girls
## 2   10    20
## 3   30    40
```

If one wants to know about defined ranges, without opening the file and inspecting, `info` helps:

```
read.xlx('survey.xlsx', info=TRUE)
```

```
## $wbsheets
## [1] "Survey1" "Survey2" "Sheet3"
##
## $rgsheets
##     blank education  students     young
##  "Sheet3" "Survey1" "Survey1" "Survey1"
##
## $rgrefs
##               blank           education            students
##  "Sheet3!$A$1:$B$2" "Survey1!$A$6:$B$8" "Survey1!$A$1:$B$3"
##               young
## "Survey1!$A$6:$A$8"
```

Skip an Arbitrary Number of Initial Rows

It is possible to skip the first n rows. The case without headers is simpler:

```
read.xlx('survey.xlsx', header.sheets=FALSE, skip=3)
```

```
##       1   2
## 6 Young Old
## 7  <NA>  20
## 8    30  40
```

"Where is the second sheet above?", you might ask.
The first three lines are skipped for all sheets. As a consequence of the skipping, `Survey2` sheet results in an empty output, so only `Survey1` sheet is returned as a single data.frame.

If headers are present, the output depends on the value of `skipafter`. Let us assume n is the number of row to skip.
If `skipafter=FALSE`, the row are *skipped above the header*, that is, the first n rows are cut away and the header will be the row immediately following those skipped.
If `skipafter=TRUE`, the row are *skipped below the header*, that is, the first sheet row is used as a header and the following n rows are cut.

What you normally want is `skipafter=FALSE`, since the data you want to import (with the related header) does not start at the first row. For example here we are intersted only to the second table of "Survey1".

```
read.xlx('survey.xlsx', sheets="Survey1", header.sheets=TRUE, skip=3, skipafter=FALSE)
```

```
##   Young Old
## 7    NA  20
## 8    30  40
```

There are situations in which there is some noisy material, perhaps some comments, between your header and the data values, that you want to trash.

Let us pretend that the first line of values in "Survey2", (10, 20), is a comment to skip:

```
read.xlx('survey.xlsx', sheets="Survey2", header.sheets=TRUE, skip=1, skipafter=TRUE)
```

```
##   EU US
## 3 30 40
```

As for blank rows, when there are skipped rows, the row numbers of the orginal sheets are kept.

Details for the Non-Causal User

Sheets are converted into data frame following other R `read.*` functions' behaviour, which means that the values of a column share a common type. Anyway in the same Excel column different cells can have different formats. Why loosing this information? It would have been possible to use a list object to model a sheet and so retain the differences, but most of the R statistic functions can effectively operate when at least at column level the formats are the same. That being said, when in a column there are different cell formats the prevailing compatible styles, will be applied to all. This will often involve the use of the R "character" type, because it is always compatible with numeric formats too.

Given this, recognised Excel styles and their R default equivalent are:

- number, accounting, currency, fraction, scientific: converted to R numeric format
- percent: converted to R numeric format, with a column "percent" attribute
- date (without time): converted to R `Date`
- datetime: converted to R `POSIXct`
- time: converted to R `POSIXct` set to "1899-12-30" or to a "HH:MM:SS.dec" string
- text: converted to R character format
- general: converted to R number format if possible (unless otherwise asked)

To get more details about number format conversion read the following section.

Rules for conversion of Excel number format

In Excel the format categories are listed in the "Format Cells" menu under "Number" tab and they will be converted as follows.

1. If the prevailing format in a column is Date/Time, in whatever (local) format, cells will be converted as an appropriate R date/time format; unless `morechar==TRUE`.

2. If the prevailing format is text, cells will be stored as R character.

3. If neither 1) nor 2) apply (e.g. prevails the Excel General format) the conversion depends on the argument `general`. If `general==character` they are stored as characters. If `general==number` (default) columns are converted to numeric format using the `na.string` character string argument, if this does not succeed, they are stored in character format.

4. If the prevailing format is percentage, 3) applies, but the attribute "percent" will be added to resulting data frames in the form of a logical vector identifying columns originally displayed in percentage format.

If `morechar==TRUE` and `general==character` everything will stored in character format. Note that for dates/times this means that the Excel internal equivalent number will be stored as a string.

If prevailing values are incompatible with some cells, NA will applied and warnings will be displayed.

Note that a single cell which is not a number and is not a string `na.string` will prevent its column to be converted as a numeric column. This may change in the future.

See how this works in practice

```
surv=read.xlx('survey.xlsx', "survey2")
```

```
surv
```

```
##   EU US
## 2 10 20
## 3 30 40
```

If the Excel user has not set a specific cell style. All cells have the Excel "general" format and the general format is mapped to R "character".

Let us assume that the user has explicitly set the values in row 2 and 3 to the Excel number style. Given the previous command

```
 surv[[1]]
```

```
## [1] 10 30
```

```
class(surv[[1]])
```

```
## [1] "numeric"
```

Numeric values 10 and 30 are converted to character to be compatible with the string "EU".

If you want to reduce the progress messages printed (perhaps because you are using a number of batch jobs), use:

```
suppressMessages( x=read.xlx('survey.xlsx') )
```

You will only get one line of +'s. I am thinking if it is convenient to totally abolish even this.

Empty objects

The general principle is: empty objects are not returned unless they are explicitly requested.

```
surv=read.xlx('survey.xlsx')
names(surv)
```

```
## [1] "Survey1" "Survey2"
```

```
read.xlx('survey.xlsx', c("survey2", "sheet3"))
```

```
## $survey2
##   EU US
## 2 10 20
## 3 30 40
##
## $sheet3
## data frame with 0 columns and 0 rows
```

```
read.xlx('survey.xlsx', "sheet3")
```

```
## NULL
```

Excel Date Oddities

Automatic Date Parsing

Let us assume that the locale of your Excel is English UK. Note that the locale is systemwide and it can be set (or identified) via the Windows Control Panel `Region and Language` dialog box, using the Formats tab. Here you can also define custom date locales.

Enter the string `20/10/2000` in cell A1. This is a legitimate British date, because in British English day comes before month and, since Excel is set to UK locale, it will recognise it as such. In fact, if you right-click on the cell and select `Format->Number` you will find that its category is "Date" and in the `Locale` drop-down "English (U.K.)" is selected.

Now write in cell A2 the string `10/20/2000`. Of course, this can't be parsed as a British date, since there is no month matching "20", while it would be OK in American English, because here day comes after month, therefore it will be interpreted therefore as a generic string. In fact, in `Format->Number` you now read that the category is "General".

You may be tempted to change the category of cell A2 to "Date" and select "English (U.S.)" in the Locale drop-down and maybe you will also select a matching type in the Type list. Unfortunately, despite changing format, the cell keeps not being recognised as a proper date. In fact, if you type `=YEAR(A1)` in another cell, the formula correctly extracts the year part of the date and gives 2000, but writing `=YEAR(A2)` gives `#VALUE!`, which signals that the string in cell A2 is not a date.

To get further insights on this issue, select again `Format->Number` for cell A1 and change the Locale drop-down from the current "English (U.K.)" to "English (U.S.)". You will see that the value displayed in the cell A1 automatically changes from `20/10/2000` to `10/20/2000`, also the formula `=YEAR(A1)` keeps working and correctly displays 2000.

Summing up: *a date should be entered respecting the locale*, after entering a proper date you can change the way it is displayed by changing the locale.

Bugs and Limitations

First of all you cannot enter dates before 01/01/1900.

Secondly, you might think that leap years are those divisible by 4, but this is wrong since years ending with "00" are leap only if divisible by 400. This means that 1900 is not because it is not divisible by 400 (and ends in double 0). Unfortunately Excel considers the non-existing date "29/2/1900" as a proper date. Some says this bug was added intentionally to keep compatibilty with Lotus 1-2-3.

Using VBA one can overcome some limitaitons.

Internal Storage Format

Whatever the locales or the components displayed, internally there is only a type denoted as *serial date-time*.

This is the number of days and possibly fraction of day since 31-th December 1899, the date origin. Therefore the value 1 represents January 1, 1900; the value 1.5 represents midday on January 1, 1900.

Due to the leap year bug said above, starting with the first of March, 1900 the serial numbers contain an extra day.

To fix this bug it is convenient to assume as the origin 30-th December 1899. This strategy is a compromise: it gives a good representation of all dates starting from 1-st March, 1900; but misrepresents the previous dates, which should be anyway unfrequent for most use-cases.

`read.xlx` follows this approach. It means that, for example, the Excel date `15/01/1900` (UK format) is converted as `14/01/1900`. Consider this fact should you import dates in the range January-February 1900.

When you enter a time, since there is no abstract time object, without a date specification. Excel assumes it is referred to the date January 1, 1900.

When you import times, this date will instead appear; also, because of the leap year bug, "12:30" becomes the POSIX value "1899-12-30 12:30:00".

How does `read.xlx` behave?

1. If a cell contains the value "Charles" and you set its format to date, the conversion value in R will be not surprisingly, NA.
2. If a date is entered not respecting the locale and you *don't try to change its format category to date*, then this is set to the "General" format category. Normally this value will be converted to a character. See ahead for exceptions.
3. If a date is entered not respecting the locale and you change the format category to date. *You have just created a monster*, since this appears as a date to those sharing your culture, the category and locale you set match the entered date, but for Excel it is not a date and it will be stored in the file as a string. When `read.xlx` finds a string pretending to be a date it imports it as a NA.

With respect to point 2) if you have set the `general` argument of `read.xlx` to "numeric", than again the value will be imported as a NA. The same will happen if the prevailing style is in the column is "numeric" (or the likes).

If you don't have control on the workbook content, the point 3) can be particularly subtle. You see apparently good looking dates, which are not such for Excel, and you have unpleasant NA surprised after import.
Next release of read.xlx will have a guess-date option to address this and get what looks like a date as a date in R too.