Log in Create Account

Tutorials

Karlijn Willems
April 3rd, 2015

R PROGRAMMING +1

Reading and Importing Excel Files into R

R Tutorial on Reading and Importing Excel Files into R. Understand how to read and import spreadsheet files using basic R and packages.

What's Excel's Connection To R?

Microsoft. It is an easily accessible tool for organizing, analyzing and storing data in tables and has a widespread use in many different application fields all over the world. It doesn't need to surprise that R has implemented some ways to read, write and manipulate Excel files (and spreadsheets in general).

This tutorial on reading and importing Excel files into R will give an overview of some of the options that exist to import Excel files and spreadsheets of different extensions to R. Both basic commands in R and dedicated packages are covered. At the same time, some of the most common problems that you can face when loading Excel files and spreadsheets into R will be addressed.

Want to dive deeper? Check out this DataCamp course on importing data with R, which has a chapter on importing Excel data.

Steps

6. There And Back Again

Step One. Your Data

What this tutorial eventually comes down to is data: you want to import it fast and easily to R. As a first step, it is therefore a good idea to have a data set on your personal computer.

There are basically two options to do this: either you have a dataset of your own or you download one from the Internet.

The latter can be somewhat challenging if you intend to analyze your data thoroughly after importing, as you will need to get a hold on a dataset that is as complete and qualitative as possible!

- The following list can be a useful help when you're not sure where to find data on the Internet. For this tutorial, make sure to save whatever data that you find on the Internet has a file extension that can be opened with Excel.
- Another option is Quandl, a search engine for numerical data. It offers millions of free and open financial, economic, and social datasets and might prove to be an easier option, especially for beginners who are not yet familiar with the field of data analysis. You can check out and use Quandl here.

Tip : if you are a beginning R programmer, you can go through our tutorial, which not only explains how to import and manipulate Quandl data sets, but also provides you with interactive exercises to slowly submerge you into Quandl.

Step Two. Prepping Your Data Set

Best Practices

Before you start thinking about how to load your Excel files and spreadsheets into R, you need to first make sure that your data is well prepared to be imported. If you would neglect to do this, you might experience problems when using the R functions that will be described in Step Three.

Here's a list of some best practices to help you to avoid any issues with reading your Excel files and spreadsheets into R:

- The first row of the spreadsheet is usually reserved for the header, while the first column is used to identify the sampling unit;
- Avoid names, values or fields with blank spaces, otherwise each word will be interpreted as a separate variable, resulting in errors that are related to the number of elements per line in your data set;
- If you want to concatenate words, do this by inserting a `.`. For example:

```
Sepal.Length
```

- Try to avoid using names that contain symbols such as `?`, `$`,`%`, `^`, `&`, `*`, `(`, `)`,`-`,`#`, `?`,`,`,`<`,`>`, `/`, `|`, `\`, `[` ,`]` ,`{`, and `}`;
- Delete any comments that you have made in your Excel file to avoid extra columns or NA's to be added to your file; and
- Make sure that any missing values in your data set are indicated with `NA`.

Saving Your Data

Make sure that your data is saved in Excel. This allows you to revisit the data later to edit, to add more data or to changing them, preserving the formulas that you maybe used to calculate the data, etc.

Microsoft Excel offers many options to save your file: besides the default extension `.xls` or `.xlsx`, you can go to the File tab, click on “Save As” and select one of the extensions that are listed as the “Save as Type” options.

The most common extensions to save datasets are `.csv` and `.txt`(as tab-delimited text file). Depending on the saving option that you choose, your data set’s fields are separated by tabs or commas. These symbols are then called the “field separator characters” of your data set.

Step Three. Preparatory Work In R

Once you have your dataset saved in Excel, you still need to set your working directory in R.

To do this, try to find out first where your working directory is set at this moment:

```
getwd()
```

Then, it could be that you want to change the path that is returned in such a way that it includes the folder where you have stored your dataset:

```
setwd("<location of your dataset>")
```

By executing this command, R now knows exactly in which folder you're working.

Step Four. Loading your Spreadsheets And Files Into R

After saving your data set in Excel and some adjusting your workspace, you can finally start with the real importing of your file into R!

This can happen in two ways: either through basic R commands or through packages. Go through these two options and discover which option is easiest and fastest for you.

Basic R Commands

The following commands are all part of R's Utils package, which is one of the core and built-in packages that contains a collection of utility functions. You will see that these basic functions focus on getting Excel spreadsheets into R, rather than the Excel files themselves.

If you are more interested in the latter, scroll just a bit further down to discover the packages that are specifically designed for this purpose.

As described in Step Two, Excel offers many options for saving your data sets and one of them is the tab-delimited text file or `*.txt` file.

If your data is saved as such, you can use one of the easiest and most general options to import your file to R: the `read.table()` function.

```
df <- read.table("<FileName>.txt",
                 header = TRUE)
```

You fill in the first argument of the `read.table()` function with the name of your text file in between `""` and its extension, while you specify in the second argument `header` if your excel file has names in the first line or top row. The `TRUE` value for the `header` argument is the default.

Remember that by executing `setwd()` R knows in which folder you're working. This means that you can also just write the file's name as an argument of the `read.table()` function without specifying the file's location, just like this:

```
df <- read.table("<FileName>.txt",
                 header = TRUE)
```

Note that the field separator character for this function is set to `""` or white space because it is meant to work for tab-delimited `.txt` files, which separate fields based on tabs. Indeed, white spaces here indicate not only one or more spaces, but also tabs, newlines or carriage returns.

set, like in the following data set:

```
1/6/12:01:03/0.50/WORST
2/16/07:42:51/0.32/ BEST
3/19/12:01:29/0.50/"EMPTY"
4/13/03:22:50/0.14/INTERMEDIATE
5/8/09:30:03/0.40/WORST
```

You can easily indicate this by adding the `sep` argument to the `read.table()` function:

script.R R Console

```r
# Read in the data
df <- .........("https://s3.amazonaws.com/assets.datacamp.com/blog_assets/scores_timed.txt",
                 header = FALSE,
                 sep="/",
                 strip.white = TRUE,
                 na.strings = "EMPTY")

# Print out `df`
print(..)
```

Solution Run

The `strip.white` argument allows you to indicate whether you want the white spaces from **unquoted** character fields stripped. It is only used when `sep` has been specified and only takes on a logical value. The `na.strings` indicates which strings should be interpreted as NA values. In this case, the string “EMPTY” is to be interpreted as an `NA` value.

You see the extra white space before the class `BEST` in the second row has been removed, that the columns are perfectly separated thanks to the

"EMPTY" in row three was replaced with NA. The decimal point did not cause any problems, since "." is the default for `read.table()`.

Lastly, since your data set did not have a header, R has provided some attributes for it, namely `V1`, `V2`, `V3`, `V4` and `V5`.

Note that if you ever come across a warning like "incomplete final line found by readTableHeader on…", try adding an End Of Line (EOL) character by moving your cursor to the end of the last line in your file and pressing enter. Save the file and try to import the file again; Normally the warning should be resolved.

Special Cases of read.table()

The following options are special cases of the versatile `read.table()` function. This means that any arguments that are used in this function can be applied in all the functions that are described in this section, and vice versa.

The variants are almost identical to `read.table()` and differ from it in three aspects only:

- The separator symbol;
- The header argument is always set at `TRUE`, which indicates that the first line of the file being read contains the header with the variable names;
- The `fill` argument is also set as `TRUE`, which means that if rows have unequal length, blank fields will be added implicitly.

Tip: for a full overview of all possible arguments that can be used in all four functions, visit the Rdocumentation page.

read.csv() and read.csv2()

`read.csv()` and `read.csv2()` have another separator symbol: for the former this is a comma, whereas the latter uses a semicolon.

Tip: find out what separator symbol is used in your `.csv` file by opening it in a text editor such as TextEdit or Notepad.

For example, this data set...

```
1;6;12:01:03;0,50;WORST
2;16;07:42:51;0,32;BEST
3;19;12:01:29;0,50;BEST
4;13;03:22:50;0,14;INTERMEDIATE
5;8;09:30:03;0,40; WORST
```

... As well as this one are `.csv` files:

```
1,6,12:01:03,0.50,WORST
2,16,07:42:51,0.32,BEST
3,19,12:01:29,0.50,BEST
4,13,03:22:50,0.14,INTERMEDIATE
5,8,09:30:03,0.40,WORST
```

Remember that both functions have the `header` and `fill` arguments set as `TRUE` by default.

To read `.csv` files that use a comma as separator symbol, you can use the `read.csv()` function, like this:

script.R R Console

Solution Run

Note that the `quote` argument denotes whether your file uses a certain symbol as quotes: in the command above, you pass `\"` or the ASCII quotation mark (“) to the `quote` argument to make sure that R takes into account the symbol that is used to quote characters.

This is especially handy for data sets that have values that look like the ones that appear in the fifth column of this example data set. You can clearly see that the double quotation mark has been used to quote the character values of the CLASS variable.

The `stringsAsFactors` argument allows you to specify whether strings should be converted to factors. The default value is set to `FALSE`.

Remember that you don’t have to type the file’s location if you have specified your working directory in R.

For files where fields are separated by a semicolon, you use the `read.csv2()` function:

script.R R Console

```
# Read in the data
df <- .........("https://s3.amazonaws.com/assets.datacamp.com/blog_assets/scores_timed2.csv",
                header = FALSE,
                col.names= c("X", "Y", "Z", "A","B"),
                fill = TRUE,
                strip.white = TRUE,
                stringsAsFactors=TRUE)

# Print out `df`
print(..)

# Inspect data types
str(..)xl
```

Solution Run

Note that the `dec` argument allows you to specify the character for the decimal mark. Make sure to specify this for your file if necessary, otherwise your values will be interpreted as separate categorical variables!

The `col.names` argument, completed with the `c()` function that concatenates column names in a vector, specifies the column names in the first row. This can be handy to use if your file doesn't have a header line, R will use the default variable names V1, V2, ..., etc. `col.names` can override this default and assign variable names. Similarly, the argument `row.names` specifies the observation names in the first column of your data set.

The command above imports the following data set:

```
1;6;12:01:03;0,50;"WORST"
2;16;07:42:51;0,32;"BEST"
3;19;12:01:29;0,50
4;13;03:22:50;0,14; INTERMEDIATE
5;8;09:30:03;0,40;"WORST"
```

and `row.names` arguments, that all fields are clearly separated, with the third unequal row filled in with a blank field, thanks to `fill = TRUE`. The added white spaces of unquoted characters are removed, just as specified in the `strip.white` argument. Lastly, strings are imported as factors because of the "TRUE" value that was provided for `stringsAsFactors` argument.

You check this last when you ran `str(df)`.

Note that the vector that you use to complete the `row.names` or `col.names` arguments needs to be of the same length of your dataset!

read.delim() and read.delim2()

Just like the `read.csv()` function, `read.delim()` and `read.delim2()` are variants of the `read.table()` function.

Remember that they are also almost identical to the `read.table()` function, except for the fact that they assume that the first line that is being read in is a header with the attribute names, while they use a tab as a separator instead of a whitespace, comma or semicolon. They also have the `fill` argument set to `TRUE`, which means that blank field will be added to rows of unequal length.

Consider the following data set:

```
1/6/12:01:03/0.50/"WORST"
2/16/07:42:51/0.32/"BEST"
3/19/12:01:29/0.50
4/13/03:22:50/0.14/ INTERMEDIATE
5/8/09:30:03/0.40/ WORST
```

script.R R Console

```
# Read in the data
df <- read.delim("https://s3.amazonaws.com/assets.datacamp.com/blog_assets/scores_timed3.txt",
                  header = FALSE,
                  sep = "/",
                  dec = ".",
                  row.names = c("O", "P", "Q"),
                  fill = TRUE,
                  strip.white = TRUE,
                  stringsAsFactors = TRUE,
                  na.strings = "EMPTY",
                  as.is = 3,
                  nrows = 5,
                  skip = 2)

# Print out `df`
print(df)
```

Run

Note that this function uses a decimal point as the decimal mark, as in: 3.1415. The `nrows` argument specifies that only five rows should be read of the original data. Lastly, the `as.is` is used to suppress factor conversion for a subset of the variables in your data, if they weren't otherwise specified: just supply the argument with a vector of indices of the columns that you don't want to convert, like in the command above, or give in a logical vector with a length equal to the number of columns that are read. If the argument is `TRUE`, factor conversion is suppressed everywhere.

Remember that factors are variables that can only contain a limited number of different values. As such, they are often called categorical variables.

You will get the following result:

```
O  3 19 12:01:29 0.50
P  4 13 03:22:50 0.14 INTERMEDIATE
Q  5  8 09:30:03 0.40        WORST
```

As the `read.delim()` is set to deal with decimal points, you can already suspect that there is another way to deal with files that have decimal commas.

The slightly different function `read.delim2()` can be used for those files:

script.R R Console

```r
# Load in the data
df <- read.delim2("https://s3.amazonaws.com/assets.datacamp.com/blog_assets/scores_timed4.txt",
                  header = FALSE,
                  sep = "\t",
                  dec = ".",
                  row.names = c("M", "N", "O"),
                  col.names= c("X", "Y", "Z", "A","B"),
                  colClasses = c(rep("integer", 2),
                                 "date",
                                 "numeric",
                                 "character"),
                  fill = TRUE,
                  strip.white = TRUE,
                  na.strings = "EMPTY",
                  skip = 2)

# Print the data
print(df)
```

Run

Note that the `read.delim2()` function uses a decimal comma as the decimal mark. An example of the use of a decimal comma is 3,1415.

Tip: for more interesting information on the decimal mark usage, illustrated with a googleVis visualization, read our blog post.

the dataset are not read into R. Secondly, colClasses allows you to specify a vector of classes for all columns of your data set. In this case, the data set has give columns, with the first two of type integer, replicating the class "integer" twice, the second of "date", the third of "numeric" and lastly, the fourth of "character". The replication of the integer type for the two first columns is indicated by the `rep` argument. Also, the separator has been defined as `"\t"`, an escape code that designates the horizontal tab.

The `read.delim2()` function that was defined above was applied to the following data set, which you also used in the exercise above:

```
ID  SCORE  TIME DECIMAL TIME    CLASS
1   6    12:01:03     0.50    WORST
2   16   07:42:51     0.32    BEST
3   19   12:01:29     0.50
4   13   03:22:50     0.14    "EMPTY"
5   8    09:30:03     0.40    WORST
```

However, you will get an error when you try to force the third column to be read in as a date. It will look like this:

```
Error in methods::as(data[[i]], colClasses[i]) :
  no method or default for coercing "character" to "date"
```

In other words, when executing the above `read.delim()` function, the time attribute is interpreted to be of the type character, which can not be converted to "date".

wasn't defined as it should have been. Only hours, minutes and seconds are given in this data set.

On top of that, they're given in a special format that isn't recognized as standard.

This is why you can first better read it in as a character, by replacing "date" by "character" in the `colClasses` argument, and then run the following command:

script.R R Console

```r
# Convert column `Z` to date
df$Z <- as.POSIXct(df$Z,format="%H:%M:%S")

# Print `df`
print(df)

# Inspect the data types of `df`
str(df)
```

Run

Note that the `as.POSIXct()` function allows you to specify your own format, in cases where you decided to use a specific time and date notation, just like in the data set above. Also note that when you inspect the result of `str(df)`, your data types will be as they need to be.

Do It Yourself

You can also write your own function to import your `.csv` and `.txt` files. Use `function()` to do this:

```
df <- function(x) read.delim(x)
```

Next, you can define a new function, `read.delim()`, that takes an `x` as an argument. You then want to make sure that `x` will be completed with a path that leads to your data set:

script.R R Console

```
# Load in the data
data <- ..("https://s3.amazonaws.com/assets.datacamp.com/blog_assets/test.txt")

# Print the data
print(....)
```

Solution Run

If you want to combine the two previous steps together to avoid the need to type the drive, the directory or the file's extension, you already define some arguments of the `read.delim()` function in the first step:

```
df <- function(x) read.delim(paste("<location of the file's folder>",
                                   x,
                                   ".txt",
                                   sep=""))
df("<Your file's name>")
```

In this case, you already concatenate the location of your file's folder, `x`, your file's extension and the field separator character in `read.delim()`. This saves you some time every time you want to import your file, because you

you already specified it in your function's definition.

Tip: don't forget to end your first argument of the function with a `/`! Read further to learn more about why this is so important.

Note that the `sep` argument indicates the separator for the function `read.delim()` and not for your data set. This is why the function above only works well with tab-delimited text files. If you want to import files with `.csv` extension, for example, you can do this with:

```
df <- function(x) read.delim(paste("<location of the file's folder>",
                                   x,
                                   ".txt",
                                   sep=""),
                             sep=",")
df("<Your file's name>")
```

Remember that the arguments that are passed to the `read.table()` function can also be used in `read.csv()`, `read.csv2()`, `read.delim()` and `read.delim2()`.

You can see one more `sep` argument added to the original function, which is an argument to the `read.delim()` function itself and not to the `paste()` function: it does not specify anything about the file's location, but is used to specify how the file should be read in.

To see this more clearly, try using this command, where the second `sep` argument has been omitted:

```
df <- function(x) read.delim(paste("<location of the file's folder>",
                                   x,
df("<Your file's name>")
```

You will most certainly get an error that relates to the file path to which you refer: the name in combination with the extension, and possibly the path, that were put into the function is separated by a comma, are most certainly the cause of this.

This example clearly illustrates the issue at hand:

```
In file(file, "rt") :
  cannot open file 'C:/Users/SomeonesName/Desktop,iris,.txt': No such file or
```

You see that instead of a white space, two commas are included into the file path. This type of path will never be understood by R. You will need to add a blackslash to the location of your file's folder, while adjusting at the same time the separator field character to a white space to solve this error.

Tip: you can make even more specifications to the original function by adding more arguments in the same way as you did for the second `sep` argument.

Packages

Not only can you import files through basic R commands, but you can also do this by loading in packages and then using the packages' functions. In practice, any R users limit themselves to using the basic R commands to save time and effort.

Remember that, if you do want to follow this approach, you need to install your packages: you need to do this right after setting your work directory, before entering any other command into the console. You can simply put the following:

```
install.packages("<name of the package>")
```

When you have gone through the installation, you can just type in the following to activate the package into your workspace:

```
library("<name of the package>")
```

To check if you already installed the package or not, type in the following:

```
any(grepl("<name of your package>",
          installed.packages()))
```

XLConnect

manipulating Microsoft Excel files from within R . You can make use of functions to create Excel workbooks, with multiple sheets if desired, and import data to them. Read in existing Excel files into R through:

```
df <- readWorksheetFromFile("<file name and extension>",
                            sheet=1,
                            startRow = 4,
                            endCol = 2)
```

The `sheet` argument specifies which sheet you exactly want to import into R. You can also add more specifications, such as `startRow` or `startCol` to indicate from which row or column the data set should be imported, or `endRow` or `endCol` to indicate the point up until where you want the data to be read in. Alternatively, the argument `region` allows you to specify a range, like `A5:B5` to indicate starting and ending rows and columns.

Alternatively, you can also load in a whole workbook with the `loadWorkbook()` function, to then read in worksheets that you desire to appear as data frames in R through `readWorksheet()`:

```
# Load in Workbook
wb <- loadWorkbook("<name and extension of your file>")
# Load in Worksheet
df <- readWorksheet(wb, sheet=1)
```

Of course, `sheet` is not the only argument that you can pass to `readWorksheet()`. If you want more information about the package or about all the arguments that you can pass to the `readWorkSheetFromFile()` function package's RDocumentation page.

xlsx Package

This is a second package that you can use to load in Excel files in R. The function to read in the files is just the same as the basic `read.table()` or its variants:

```
df <- read.xlsx("<name and extension of your file>",
                sheetIndex = 1)
```

Note that it is necessary to add a sheet name or a sheet index to this function. In the example above, the first sheet of the Excel file was assigned.

If you have a bigger data set, you might get better performance when using the `read.xlsx2()` function:

```
df <- read.xlsx2("<name and extension of your file>",
                 sheetIndex = 1,
                 startRow=2,
                 colIndex = 2)
```

Fun fact: according to the package information, the function achieves a performance of an order of magnitude faster on sheets with 100,000 cells or more. This is because this function does more work in Java.

Note that the command above is the exact same that you can use in the `readWorkSheetFromFile()` from the XLConnect package and that it specifies that you start reading the data set from the second row onwards.

extract.

Just like XLConnect, the xlsx package can do a lot more than just reading data: it can also be used to write data frames to Excel workbooks and to manipulate the data further into those files. If you would also like to write a data frame to an Excel workbook, you can just use `write.xlsx()` and `write.xlsx2()`.

Note the analogy with `read.xlsx()` and `read.xlsx2()`!

For example:

```
write.xlsx(df,
           "df.xlsx",
           sheetName="Data Frame")
```

The function requires you first to specify what data frame you want to export. In the second argument, you specify the name of the file that you are outputting.

Note that this file will appear in the folder that you designated as your working directory.

If, however, you want to write the data frame to a file that already exists, you can execute the following command:

```
write.xlsx(df,
           "<name and extension of your existing file>",
```

Note that, in addition to changing the name of the output file, you also add the argument `append` to indicate that the data frame sheet should be added to the given file.

For more details on this package and its functions, go to this page.

gdata Package

This package provides another cross-platform solution to load in Excel files into R. It contains various tools for data manipulation, among which the `read.xls()` function, which is used as follows:

```
df <- read.xls("<name of your file.xls>",
               perl="<location of perl.exe file on your pc")
```

The `read.xls()` function translates the named Excel File into a temporary `.csv` or `.tab` file, making use of Perl in the process.

Perl is a programming language and stands for “Practical Extraction and Report Language”. It comes standard on Linux and MAC OS X. If you are using Windows and you do not have it installed on your computer, you can download ActiveState Perl here. For more information on how to fill in the `perl` argument, visit this page.

Note that you don’t need to know how to use Perl, you just need to be able to retrieve its location on your computer! :)

spreadsheets by the reference to .xls, but it also accepts .xlsx files as input.

Note that you actually need to specify the exact Perl path in the perl argument to execute this command without prompting any errors, only if the working version of Perl is not in the executable search path. In other words, when the read.xls() funtion is executed, R searches the path to the Excel file and hopes to find Perl on its way.

If this is not the case, R will return an error. A possible completion of the perlargument can look like this, for example:

```
df <- read.xls("iris.xls",
               sheet=1,
               perl="C:/Perl/bin/perl.exe")
```

This package also offers other functions, such as xls2sep() and its wrappers xls2csv(), xls2tab() and xls2tsv() to return temporary files in .csv, .tab, .tsv files or any other specified format, respectively.

These functions work exactly the same as read.xls():

```
df <- xls2csv("<name of your file>.xls",
              sheet = 1,
              na.strings = "EMPTY",
              perl="<location of Perl>")
```

The output of this function, df, will contain the temporary .csv file of the first sheet of the .xls or .xlsx file with stringS “EMPTY” defined as NA previous functions that is fit to read in files with the .csv extension, like read.csv():

```
df <- read.csv(df)
```

Note that any additional arguments for `read.xls()`, `xls2sep()` and its wrappers are the same as the ones that you use for `read.table()`. The defaults of these arguments as set as the ones for `read.csv()`: the `header` and `fill` arguments set as `TRUE` by default and the separator symbol is ",".

If you want to make sure which file formats are supported by the `read.xls()` function, you can use the `xlsFormats()` function:

```
xlsFormats(perl="<location of Perl>")
```

readxl Package

A more recent entrant into the ranks of the `.xlsx` and `.xls` data importing libraries is Hadley Wickham's 2016 `readxl` package. By leveraging R's established `RapidXML` C++ library, `readxl` can increase both the speed and the ease of loading excel documents into R.

Since no external code libraries (e.g.: `java jdk` or `ActiveState PERL`) are required, the setup is extremely simple.

Additionally, as the `readxl` package is already bundled into the increasingly foundational `tidyverse` package, the more recent generations of R users may be delighted to discover that they have already installed everything they need to start effortlessly pulling in excel docs!

libraries that includes readxl), or you have chosen to install the solitary readxl package from CRAN or GitHub, you will still need to explcitly load the readxl library in by name - since it is not one of the core tidyverse packages.

Easy enough: at the beginning of your script, simply add library(readxl) to the list of libraries you are loading.

readxl's functions and arguments are simple and straightforward. To read in the first tab of your excel sheet, simply enclose your file name inside the read_excel() function.

```
df <- read_excel("<name and extension of your file>")
```

In other words, the default is to read the first sheet(tab) in the specified workbook. If your workbook is a little more complicated than this, you can crack it open and list the sheet names with the following excel_sheets function:

```
excel_sheets("<name and extension of your file>")
```

From there, you can then choose which sheet to read with the sheet argument: either referencing the sheet's name or its index (number).

References to sheet names are direct and therefore do require quotes:

```
read_excel("<name and extension of your file>",
           sheet="Sheet 3")
```

Sheet indexing starts at 1, so alternatively, you could load in the third tab in with the following code:

```
read_excel("<name and extension of your file>",
          sheet=3)
```

In the `read_excel` function, if the `col_names` argument is left to its default value of `True`, you will import the first line of the worksheet as the header names. In line with `tibble` and `tidyverse` standards, the `readxl` column header names are formed exactly as they were written in Excel.

This results in behaviour that is much more in line with the expectations of Excel and tidy data users.

If you want to convert column names to classic Base R valid identifiers, base R's `make.names()` is able to quickly perform the necessary conversions. Leading numbers and symbols will be prefixed or replaced with `X`'s and spaces will be replaced with `.`'s.

Alternatively, if you wish to skip using header specified column-names and instead "number columns sequentially from X1 to Xn", then set this argument to false: i.e. `col_names = FALSE`

Leaving the `col_types` argument in its default state will cause types to be automatically registered when `read_excel()` samples the first 10 rows and assigns each column to the most applicable class. As with `read.table`'s `colClasses` argument that you've seen earlier, you can also manually classify column types on entry.

column, however, this time be sure to use the following classification options of “blank”, “numeric”, “date”, or “text”.

For example, if you want to set a three column excel sheet to contain the data as dates in the first column, characters in the second, and numeric values in the third, you would need the following lines of code:

```
read_excel("<name and extension of your file>",
           col_types = c("date", "numeric", "text"))
```

While this is easy enough for tall datasets, with wider dataframes you want to transform only a few column types after the import using `as.character` or `as.numeric` type mutations.

If you wish to avoid all issues from the beginning, and bring all your excel data into R in the most encompassing way possible, you can simply specify each column to be cast as characters. For a ten-column sheet this would look like the following:

```
read_excel("<name and extension of your file>",
           col_types = rep("text", 10))
```

For the final of the most useful additional arguments available in `read_excel`, if you wish to skip rows before setting column names, there is the `skip` argument. This works exceptionally well for dealing with those intricately crafted database reports you enjoy so much.

rows of report generation details, and the column headers in the sixth row. Getting this imported quickly and tidily into R requires only the following code:

```
read_excel("<name and extension of your file>",
           skip = 5)
```

For more details on this package and its functions, please see this page.

Step Five. Final Checkup

After executing the command to read in the file in which your data set is stored, you might want to check one last time to see if you imported the file correctly.

Remember to type in the following command to check the attributes' data types of your data set:

```
str("<name of the variable in which you stored your data>")
```

Alternatively, you can also type in:

```
head("<name of the variable in which you stored your data>")
```

By executing this command, you will get to see the first rows of your data frame. This will allow you to check if the data set's fields were correctly separated, if you didn't forget to specify or indicate the header, etc.

data frame rows you want to return, like in: `head(df, 5)` to return the first five lines of the data frame `df`.

Step Six. There And Back Again

Importing your files is only one small but essential step in your endeavours with R. From this point, you are ready to start analyzing, manipulating or visualizing the imported data.

Do you want to continue already and get started with the data of your newly imported Excel file? Check out our tuturials for beginners on histograms and machine learning.

This tutorial was written in collaboration with Jens Leerssen, Data Quality Analyst with a passion for resolving data quality issues at scale in large, documentation sparse environments.

▲
0

Subscribe to RSS

About Terms Privacy