

(C) (cc by-sa) Wouter van Atteveltdt, file generated mei 24 2016

Note on the data used in this howto: This data can be downloaded from <http://piketty.pse.ens.fr/files/capital21c/en/xls/>, but the excel format is a bit difficult to parse as it is meant to be human readable, with multiple header rows etc. For that reason, I've extracted csv files for some interesting tables that I've uploaded to <https://github.com/vanatteveldt/learningr/tree/master/data>. If you're accessing this tutorial from the github project, these files should be in your 'data' sub folder automatically.

Organizing data in R

This hands-on demonstrates reading, writing, and manipulating data in R. As before, we will continue using the data from Piketty's 'Capital in the 21st Century'

```
income = read.csv("data/income_toppercentile.csv")
```

Saving and loading data

So far, we've used the `read.csv` command to read data from a CSV file. As can be guessed, there is also a `write.csv` command that writes data into a CSV file:

```
write.csv(income, file="test.csv")
test = read.csv("test.csv")
head(test)
```

```
##   X Year Canada Australia New.Zealand Denmark Italy Holland Spain France
## 1 1 1900      NA         NA           NA      NA      NA      NA      NA      NA
## 2 2 1901      NA         NA           NA      NA      NA      NA      NA      NA
## 3 3 1902      NA         NA           NA      NA      NA      NA      NA      NA
## 4 4 1903      NA         NA           NA    0.162    NA      NA      NA      NA
## 5 5 1904      NA         NA           NA      NA      NA      NA      NA      NA
## 6 6 1905      NA         NA           NA      NA      NA      NA      NA      NA
##   US
## 1 NA
## 2 NA
## 3 NA
## 4 NA
## 5 NA
## 6 NA
```

A new column was created because by default `write.csv` also writes the row numbers (you can check this by opening `test.csv` in excel). Since this row number column has no header, it is given the variable name `X`. You can suppress this by adding `row.names=F` to the `write.csv` function:

```
write.csv(income, file="test.csv", row.names=F)
```

On european computers, excel produces (and expects) csv files to be delimited with semicolons rather than commas by default, using the comma as a decimal separator (instead of period). To facilitate this, R provides a pair of functions `read.csv2`/`write.csv2` that use this format.

If you open a CSV file using the wrong function, you will only see a single column with all the values in it. For example, if we use `read.csv2` to open the file we just created we get the following:

```
d = read.csv2("test.csv")
head(d)
```

```
##   Year.Canada.Australia.New.Zealand.Denmark.Italy.Holland.Spain.France.US
## 1          1900,NA,NA,NA,NA,NA,NA,NA,NA,NA,NA
## 2          1901,NA,NA,NA,NA,NA,NA,NA,NA,NA,NA
## 3          1902,NA,NA,NA,NA,NA,NA,NA,NA,NA,NA
## 4          1903,NA,NA,NA,0.162,NA,NA,NA,NA,NA,NA
## 5          1904,NA,NA,NA,NA,NA,NA,NA,NA,NA,NA
## 6          1905,NA,NA,NA,NA,NA,NA,NA,NA,NA,NA
```

The bottom line is: when using CSV data, always check your results, and use the ‘European’ version of the commands when appropriate.

Apart from writing csv files, R can also write to a native file format, which has the advantage of correctly storing all types of data (including numbers and date columns) and of storing multiple variables in one file.

For example, the following code stores the `income` and a new `x` variable in a file called `mydata.rdata`:

```
x = 12
save(income, x, file="mydata.rdata")
```

Now, you can clear the data from your environment, using the Clear button in RStudio or by issuing the somewhat cryptic command `rm(list=ls())`

```
rm(list=ls())
head(income)
```

```
## Error in head(income): object 'income' not found
```

And if you load the file, the variables will appear again:

```
load("mydata.rdata")
head(income)
```

```
##   Year Canada Australia New.Zealand Denmark Italy Holland Spain France US
## 1 1900      NA        NA           NA      NA      NA      NA      NA      NA NA
## 2 1901      NA        NA           NA      NA      NA      NA      NA      NA NA
## 3 1902      NA        NA           NA      NA      NA      NA      NA      NA NA
## 4 1903      NA        NA           NA    0.162    NA      NA      NA      NA NA
## 5 1904      NA        NA           NA      NA      NA      NA      NA      NA NA
## 6 1905      NA        NA           NA      NA      NA      NA      NA      NA NA
```

Note that you do not load the file into a specific variable, as the file can contain multiple variables. The load command will automatically create those variables with their original names.

Subsetting data

The data we have downloaded into `income` contains income series from 1900 to 2010 for a number of countries. We can use hard brackets `[rows, columns]` to subset this dataset, for example to select only the first 10 rows or to only select the US and French data.

```
income[1:10, ]
```

```
##      Year Canada Australia New.Zealand Denmark Italy Holland Spain France US
## 1  1900      NA        NA           NA      NA      NA      NA      NA      NA NA
## 2  1901      NA        NA           NA      NA      NA      NA      NA      NA NA
## 3  1902      NA        NA           NA      NA      NA      NA      NA      NA NA
## 4  1903      NA        NA           NA    0.162    NA      NA      NA      NA NA
## 5  1904      NA        NA           NA      NA      NA      NA      NA      NA NA
## 6  1905      NA        NA           NA      NA      NA      NA      NA      NA NA
## 7  1906      NA        NA           NA      NA      NA      NA      NA      NA NA
## 8  1907      NA        NA           NA      NA      NA      NA      NA      NA NA
## 9  1908      NA        NA           NA    0.165    NA      NA      NA      NA NA
## 10 1909      NA        NA           NA      NA      NA      NA      NA      NA NA
```

```
subset = income[, c("US", "France")]
head(subset)
```

```
##      US France
## 1 NA      NA
## 2 NA      NA
## 3 NA      NA
## 4 NA      NA
## 5 NA      NA
## 6 NA      NA
```

A more common use case is that we want to select based on specific criteria. Suppose that we are now only interested in the series for the US, and France since 1945. We can place an expression in the rows selector to subset the data like that:

```
subset = income[income$Year > 1945, c("Year", "US", "France")]
head(subset)
```

```
##      Year      US France
## 47 1946 0.133 0.092
## 48 1947 0.120 0.092
## 49 1948 0.122 0.088
## 50 1949 0.117 0.090
## 51 1950 0.128 0.090
## 52 1951 0.118 0.090
```

Calculating columns

We saw earlier that you can store the result of a calculation in a new variable. You can also create a new column by storing the result of a calculation in a column. For example, we could create a column for the average of US and French inequality:

```
subset$average = (subset$US + subset$France) / 2
head(subset)
```

```
##      Year      US France average
## 47 1946 0.133 0.092 0.1125
## 48 1947 0.120 0.092 0.1060
## 49 1948 0.122 0.088 0.1050
## 50 1949 0.117 0.090 0.1035
## 51 1950 0.128 0.090 0.1090
## 52 1951 0.118 0.090 0.1040
```

It is also possible to replace part of a column. For example, we can set the average to NA when the French value is lower than 0.09 like so:

```
subset$average[subset$France < 0.09] = NA
head(subset)
```

```
##      Year      US France average
## 47 1946 0.133 0.092 0.1125
## 48 1947 0.120 0.092 0.1060
## 49 1948 0.122 0.088      NA
## 50 1949 0.117 0.090 0.1035
## 51 1950 0.128 0.090 0.1090
## 52 1951 0.118 0.090 0.1040
```

What you are doing there is in fact assigning NA to a subset of the column, selected using the France column. Becoming good at R for a large part means becoming good at using the subsetting and assignment operations, so take some time to understand and play around with this code.

Dealing with Missing Values

Finally, a useful function is `is.na`. This function is true when it's argument is NA (i.e., missing):

```
is.na(subset$average)
```

```
## [1] FALSE FALSE TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [12] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [23] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
## [34] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
## [45] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
## [56] TRUE TRUE TRUE TRUE TRUE TRUE TRUE FALSE TRUE TRUE TRUE
```

As you can see, it is true for the third row and for most rows past the 23d. In fact, an expression like `subset$average > 3` also returns such a vector of logical values:

```
subset$US > .11
```

```
## [1] TRUE TRUE TRUE TRUE TRUE TRUE FALSE FALSE FALSE TRUE FALSE
## [12] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [23] TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [34] FALSE FALSE FALSE FALSE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
## [45] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
## [56] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
```

This result is TRUE for those years where the income inequality in the US is larger than .11. Just as we can use `subset$France < 0.09` to selectively replace certain cells, we can do so with `is.na`:

```
subset$average[is.na(subset$average)] = 0
head(subset)
```

```
##      Year      US France average
## 47 1946 0.133 0.092 0.1125
## 48 1947 0.120 0.092 0.1060
## 49 1948 0.122 0.088 0.0000
## 50 1949 0.117 0.090 0.1035
## 51 1950 0.128 0.090 0.1090
## 52 1951 0.118 0.090 0.1040
```

This command tells R to replace every cell in the average column where the average is missing with zero. Since sometimes NA values are really zero, this is quite a useful command. We can also use this to remove NA rows, similar to the `na.omit` command used earlier but more flexible. Let's first introduce our NA's again:

```
subset$average[subset$France < 0.09] = NA
head(subset)
```

```
##      Year      US France average
## 47 1946 0.133 0.092 0.1125
## 48 1947 0.120 0.092 0.1060
## 49 1948 0.122 0.088      NA
## 50 1949 0.117 0.090 0.1035
## 51 1950 0.128 0.090 0.1090
## 52 1951 0.118 0.090 0.1040
```

And now use `!is.na` to select certain rows in the data frame (an exclamation mark (read as NOT) inverts a selection)

```
subset.nomissing = subset[!is.na(subset$average), ]
head(subset.nomissing)
```

```
##      Year      US France average
## 47 1946 0.133 0.092 0.1125
## 48 1947 0.120 0.092 0.1060
## 50 1949 0.117 0.090 0.1035
## 51 1950 0.128 0.090 0.1090
## 52 1951 0.118 0.090 0.1040
## 53 1952 0.108 0.092 0.1000
```

As you can see, row 49 is gone. Note the trailing comma in the subset command. Although we only want to select on rows (and not on columns), we still need to place a comma after the row selection to complete the `[rows, columns]` pattern.

In fact, you can also use selections on a whole data frame, allowing you to replace all values under a certain condition.

```
subset[subset < .11] = NA
head(subset, n=10)
```

```
##      Year      US France average
## 47 1946 0.133      NA 0.1125
## 48 1947 0.120      NA      NA
## 49 1948 0.122      NA      NA
## 50 1949 0.117      NA      NA
## 51 1950 0.128      NA      NA
## 52 1951 0.118      NA      NA
## 53 1952      NA      NA      NA
## 54 1953      NA      NA      NA
## 55 1954      NA      NA      NA
## 56 1955 0.111      NA      NA
```

Note that here the trailing comma is not given since the selection is based on the whole data set, not just on certain rows. Similarly, the `is.na` function can be used to globally replace NA values in a data frame:

```
subset[is.na(subset)] = 0
head(subset, n=10)
```

```
##      Year      US France average
## 47 1946 0.133      0 0.1125
## 48 1947 0.120      0 0.0000
## 49 1948 0.122      0 0.0000
## 50 1949 0.117      0 0.0000
## 51 1950 0.128      0 0.0000
## 52 1951 0.118      0 0.0000
## 53 1952 0.000      0 0.0000
## 54 1953 0.000      0 0.0000
## 55 1954 0.000      0 0.0000
## 56 1955 0.111      0 0.0000
```

Good practice: self-contained scripts

Using R is programming, and one of the most important parts of programming is managing your source code. An important thing to realize is that your code will be written only once, but read many times over. Spending twice as much time to make the code well organized and more readable might feel like wasting time, but you (or your colleagues/students) will be very happy when you are reading it again. Especially since in research code is often left alone for a number of months until it is time to review an article, it is very important to make sure that you (and ideally: the readers/reviewers of the article) can understand the code.

Although there are no simple rules for writing readable code, and sometimes what is readable to one is quite cryptic to the other. However, here are three tips that I can offer and that I expect you to incorporate in your assignments:

1. Use descriptive variable names. Use `income` (or better: `income.top.percent`) rather than `i`.
2. Use comments where needed, especially to explain decisions, assumptions, and possible problems. In R, every line starting with `#` is a comment, i.e. the line is completely skipped by R.

3. Often, when doing an analysis you're not quite sure where you are going to end up, so you write a lot of code that turns out not to be needed. When your analysis is done, take a moment to reorganize the code, remove redundancies, et cetera. It is often best to just start a new file and copy paste the relevant bits (add comments where needed). Assume that your code will also be reviewed, even if it is not, because you are sure to read it again later and wonder why/how you did certain things.
4. Finally, try to write what I term 'self contained scripts'. The script should start with some kind of data gathering commands such as `download.file` or `read.csv`, and end with your analyses. You should be able to clear your environment and run the code from top to bottom and arrive at the same results. In fact, when cleaning up my code I often do just that: clean up part of the code, clear all, re-run, and check the results. This is also important for reproducibility, as being able to run the whole code and get the same results is the only guarantee that that code in fact produced these results.

We will come across some tools to make these things easier such as defining your own functions and working with knitr, but the most important thing is to accept the your code is part of your product and you should take the time to polish it a bit.