R: Core Skills for Reproducible Research

Course Manual with Practical Exercises

Maja Založnik

Contents

1 Introduction

1.1 Blurb

This short course covers the core skills required for a budding R user to develop a strong foundation for data analysis in the RStudio environment. Within the framework of a reproducible research workflow we will cover importing and cleaning data, efficient coding practices, writing your own functions and using the powerful `dplyr` data manipulation tools.

1.2 Key Topics

- Reproducible Research
- R Studio and project management
- Importing and cleaning data
- Good coding practices in R
- standard control structures
- Vectorisation and `apply` functions
- Writing your own funcitons
- Data manipulation with `dplyr`
- Piping/chaining commands

1.3 Course information

Intended audience Anyone interested in quantitative data analysis using open source tools.

Prior knowledge Knowledge of R (as covered in R: An introduction).

Resources Course handbook

Software RStudio & R 3.1.2

Format Presentation with practical exercises

Where next? R:

2 Reproducible Research

Reproducible reseach means making the data and the code of our analysis available in a way that is sufficient and easy for an independent researcher to recreate our findings.

This is the golden standard of scientific inquiry, and is increasinlgy and rightly becoming a requirement in academic publishing, and by funding bodies.

It is also a way of establishing better working habits, reduce the potential for error, develop a more streamlined research process, and make for easier collaboration.

Reproducible reseach does take a bit of upfront investment in learning the tools and setting up your workflow. Luckily RStudio has integrated many of the tools required in one platform, making it easier than ever to

2.1 Why?

- Reinhart Rogoff Excel spreadsheet

Document everything! This means never running any code from the command prompt, always writing it into a script file and running it from there.

3 Set-up

3.1 RStudio

3.2 Project management

A crucial requirement for conducting reproducible research, and one that has to be carefully considered before you embark on your analysis, is your plan on how the data, code and outputs will be organised. The project management structure proposed here is just a suggestion, and you should adapt it to your specific needs, but it is highly recommended that you stick to one such system consistently, instead of comming up with 'ad hoc' solutions for every new project.

RStudio makes it extremely easy to divide your work into separate projects, allowing you to neatly organize and acces your work.

3.3 Literate programming

3.3.1 Consistent coding style e.g.:

- Google's R Style Guide
- Hadley Wickham's Style Guide

3.3.2 File formats

Human readability of data files and outputs. Future-proof. .txt files

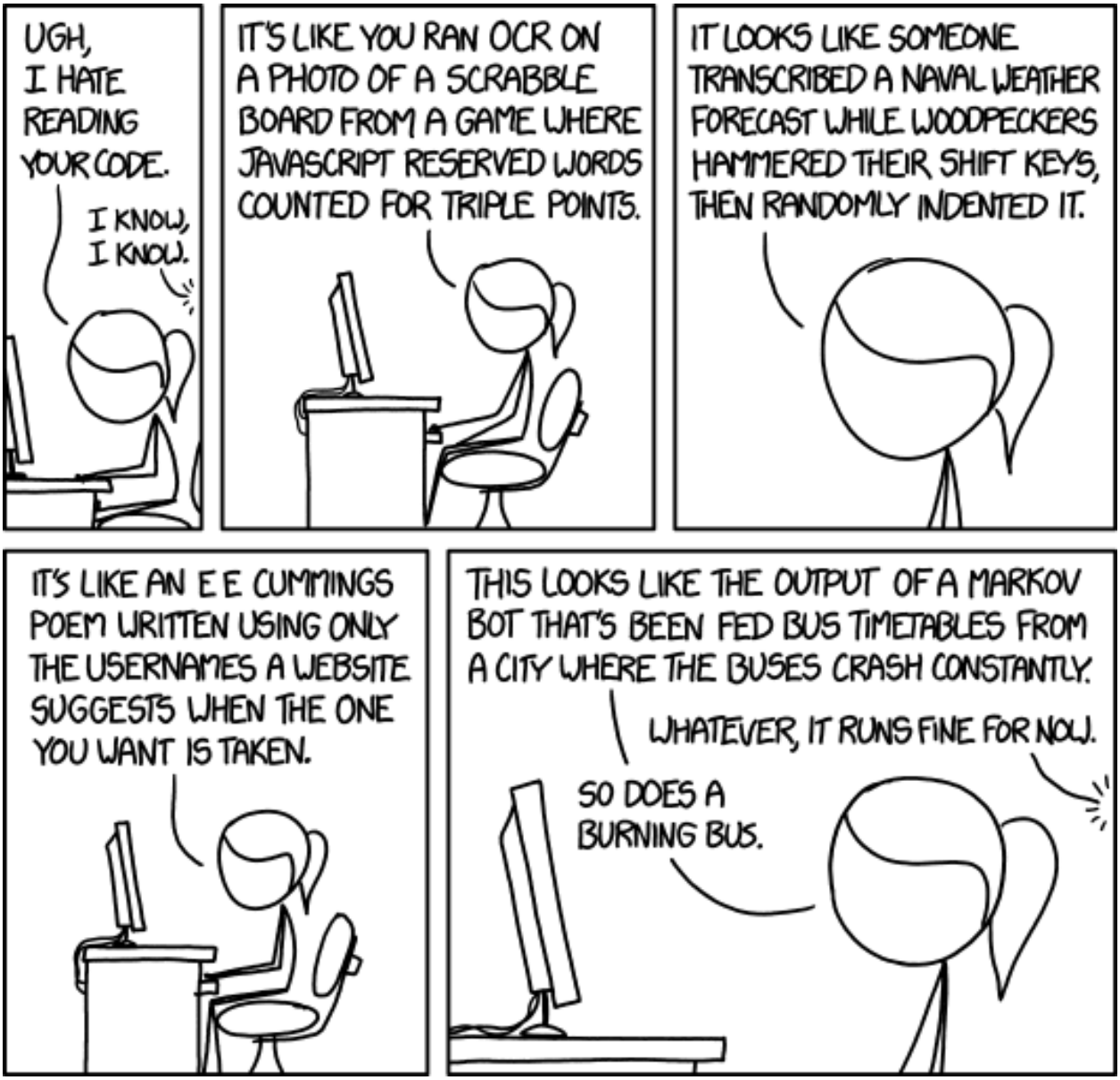

Figure 1: Code Quality part 2. (https://xkcd.com/1695/)

3.3.3 Commenting

3.4 PRACTICAL: new R project

- personalise RStudio settings (don't save .Rdata etc)
- new project folder with subfolders (data, figures, scripts)
- new Rproject

4 Workflow

> *Data scientists, according to interviews and expert estimates, spend from 50 percent to 80 percent of their time mired in this more mundane labor of collecting and preparing unruly digital data, before it can be explored for useful nuggets.*
>
> source: NY Times

4.1 Importing data

Regardless of whether your data is stored locally or downloaded from the web, you should never manipulate the original data directly. This is cruical for the integrety of your reproducible research process.

R has utilities for importing data from a wide variety of sources including proprietary formats. Ideally you want to be working with .csv files, as they are the cleanest and least problematic to import, but often you have no choice in the matter. In the practical we will import .csv, .xls and .sav files, including downloading and unzipping them. Here is a list of some common formats and the packages used for importing them, refer to the help pages for more details:

- comma separated values – `read.csv()`
- tab-delimited text file – `read.table()`
- other delimited files – `read.delim()`
- Minitab – `read.mtb()` from `library(foreign)`
- SPSS – `read.spss()` from `library(foreign)`
- Stata – `read.dta()` from `library(foreign)`
- Excel – `read.xls()` from `require(gdata)`
- Excel – `loadWorkbook()` from `library(XLConnect)`

The basic import functions of the `read.table()` family all have a `nrows` argument, which is particularly useful if you do not know the structure of the data and are dealing with a large fine. In which case it is recommended you try a test import with e.g. `nrows=10`, and check the result before attempting to import the full file.

For a more comprehensive list of possible imput formats see this tutorial: https://www.datacamp.com/community/tutorials/r-data-import-tutorial

We will store all our data files in the `data` folder of our project, from where they will be imported into R. This means the original files remain *untouched* by the data analysis and should never be overwritten as the result of your analysis.

While you might find it easier to simply download a file into your folder, this poses the problem of loosing track of where the data was sourced from. It is therefore highly recommended you download the data programmatically if possible, and if not, that you use comments within the code to describe the source of the files. For example the `pop2010.csv` file we downloaded in the first practical should have been downloaded directly from within R, by doing it manually we are reducing the reproducibility of our project. We must therefore make sure we note the origin and date we accessed the data in our code!

4.2 Data tidying

Tidy datasets are all alike but every messy dataset is messy in its own way. – Hadley Wickham

A great deal of data tidying can be done manually with the base R funcitons. Additionally there are several packages available with more specific functions. In this course we will use the `tidyr` package by Hadley Wickham, which is particularly well integrated with the `dplyr` package we will be using in the second part of this course.

The underlying principle of the `tidyr` package is *tidy data*, which must satisfy the following three principles:

1. Each variable forms a column.
2. Each observation forms a row.
3. Each type of observational unit forms a table.

Source: H. Wickham (2014) *Tidy Data* (available: http://vita.had.co.nz/papers/tidy-data.pdf)

This may seem trivial, but it is in fact common to encounter data that does not conform to these principles. The four workhorse funcitons of `tidyr` that should solve all your data tidying needs are:

- `spread()`
- `gather()`
- `separate()`
- `unite()`

4.2.1 spread()

Below we can see an example of a messy table, since each observation is in fact represented in two rows. Third column in fact contains variable names (`density` and `population`), while the fourth column contains their values.

```
##     country year        key       value
## 1    Norway 2010 population  4891300.00
## 2    Norway 2010    density       16.07
## 3    Norway 2050 population  6364008.00
## 4    Norway 2050    density       20.91
## 5  Slovenia 2010 population  2003136.00
## 6  Slovenia 2010    density       99.41
## 7  Slovenia 2050 population  1596947.00
## 8  Slovenia 2050    density       79.25
## 9        UK 2010 population 62348447.00
## 10       UK 2010    density      257.71
## 11       UK 2050 population 71153797.00
## 12       UK 2050    density      294.11
```

we can using `spread()` we can tidy this layout:

```
tidy.02 <- spread(messy.02, key, value)
tidy.02
```

```
##    country year density population
## 1   Norway 2010   16.07    4891300
```

```
## 2   Norway 2050   20.91    6364008
## 3 Slovenia 2010   99.41    2003136
## 4 Slovenia 2050   79.25    1596947
## 5       UK 2010  257.71   62348447
## 6       UK 2050  294.11   71153797
```

The sytnax for `spread()` takes the following form:

```
spread(data, key, value)
```

The *key–value* pair is the underlying logic of the tidy data table. We can decompose the data into a collection of key–value pairs such as this:

Key : Value

```
Country: Norway
Country: Slovenia
Country: UK

Year: 2010
Year: 2050

Population: 4891300
Population: 2003136
...

Density: 16.07489
Density: 20.91484
...
```

In a tidy data table each cell contains a *value* and the *keys* are the column names.

4.2.2 gather()

Here is another messy table:

```
messy.01
```

```
##    country    X2010    X2050
## 1   Norway  4891300  6364008
## 2 Slovenia  2003136  1596947
## 3       UK 62348447 71153797
```

Now we have three variables: the country, which is in the first column, the year, which is across the header row (representing the *keys*), and the population (representing the *values*), which is in the second and third columns. Using `gather()` we can tidy up the table, so that now each of the three variables has its own column, and each row is an observation:

The sytnax for `gather()` takes the following form:

```
gather(data, key, value, ...)
```

where the `...` represents the columns we want to gather, in our case columns 2 and 3. The key and value arguments are the *names* of the two new variables, or columns we are creating: the *key* is currently in the column names of columns two and three - so we want it to become `year`, and the *values* are in the cells of those two columns, so we want it to become `population`.

```
tidy.01 <- gather(messy.01, year, population, 2:3)
tidy.01
```

```
##    country year population
## 1   Norway 2010    4891300
## 2 Slovenia 2010    2003136
## 3       UK 2010   62348447
## 4   Norway 2050    6364008
## 5 Slovenia 2050    1596947
## 6       UK 2050   71153797
```

4.2.3 separate() and unite()

Separeate and unite are straightforward helper funcitons for the reshaping done by gather and spread. The following table for example requires spreading, but the `double.key` variable contains both *values* (years) and *keys* (population and density):

```
##     country      double.key       value
## 1    Norway 2010_population  4891300.00
## 2    Norway    2010_density       16.07
## 3    Norway 2050_population  6364008.00
## 4    Norway    2050_density       20.91
## 5  Slovenia 2010_population  2003136.00
## 6  Slovenia    2010_density       99.41
## 7  Slovenia 2050_population  1596947.00
## 8  Slovenia    2050_density       79.25
## 9        UK 2010_population 62348447.00
## 10       UK    2010_density      257.71
## 11       UK 2050_population 71153797.00
## 12       UK    2050_density      294.11
```

These are separated simply by the following code into `year` and `key`, which can then be used to reshape the table as we did above.

```
##     country year        key       value
## 1    Norway 2010 population  4891300.00
## 2    Norway 2010    density       16.07
## 3    Norway 2050 population  6364008.00
## 4    Norway 2050    density       20.91
## 5  Slovenia 2010 population  2003136.00
## 6  Slovenia 2010    density       99.41
## 7  Slovenia 2050 population  1596947.00
## 8  Slovenia 2050    density       79.25
## 9        UK 2010 population 62348447.00
## 10       UK 2010    density      257.71
## 11       UK 2050 population 71153797.00
## 12       UK 2050    density      294.11
```

The function `unite` is the inverse of `separate`, and merges the values of selected columns into a new single column. In both cases you can change the separator using the `sep=` argument.

```
##     country              new.double.key       value
## 1    Norway population in the year 2010  4891300.00
## 2    Norway    density in the year 2010       16.07
## 3    Norway population in the year 2050  6364008.00
## 4    Norway    density in the year 2050       20.91
## 5  Slovenia population in the year 2010  2003136.00
## 6  Slovenia    density in the year 2010       99.41
## 7  Slovenia population in the year 2050  1596947.00
## 8  Slovenia    density in the year 2050       79.25
## 9        UK population in the year 2010 62348447.00
## 10       UK    density in the year 2010      257.71
## 11       UK population in the year 2050 71153797.00
## 12       UK    density in the year 2050      294.11
```

4.3 PRACTICAL: Import and clean some data

4.3.1 Downloading and importing data

First create a new `.R` file in your `scripts` folder or equivalent. Naming it something like `01-DataImport.R` will make your project management easier in the long run, but feel free to set up your own file naming system – but try to stick with it!

It is good practice to establish a header system for all your script files, such as the one below. The `#` lines are also a good way of making the code file structure easy to understand. Following the Google's R Style Guide rule "The maximum line length is 80 characters.", a nice little trick is to make these separators 80 hashtags long, which gives you a nice visual reference for when your code gets too wide.

```
################################################################################
## DATA IMPORT AND CLEANUP
################################################################################
## 1.1  Import a .csv file
## 1.2  Download and import an Excel file
## 1.3  Download, unzip and import a .dat file
################################################################################
```

Make sure your working directory is at the top of your project folder using `getwd()`. We are going to use the `read.csv()` funciton to import the data from the `pop2010.csv` file in the `data` folder. But just to be safe, we will first do a test run, importing only 10 rows, so we can inspect the result before importing the whole table:

```
## 1.1  Import a .csv file
################################################################################
# don't forget to note the source of the data!
# downloaded manually from https://github.com/majazaloznik/RepResCoreSkillsR/
# blob/master/data/pop2010.csv?raw=true" on 20.6.2016
getwd()
```

```
## [1] "C:/Users/sfos0247/Dropbox/XtraWork/R stuff/RepResCoreSkillsR"
```

```
# test run
population2010 <- read.csv("data/pop2010.csv", nrows=10)
population2010
```

```
##    AGE AREA_KM2                 NAME    POP SEX FIPS time
## 1    0      180                Aruba    660   1   AA 2010
## 2    0      180                Aruba    653   2   AA 2010
## 3    0      443  Antigua and Barbuda    720   1   AC 2010
## 4    0      443  Antigua and Barbuda    688   2   AC 2010
## 5    0    83600 United Arab Emirates  40770   1   AE 2010
## 6    0    83600 United Arab Emirates  38987   2   AE 2010
## 7    0   652230          Afghanistan 534585   1   AF 2010
## 8    0   652230          Afghanistan 516673   2   AF 2010
## 9    0  2381741              Algeria 434735   1   AG 2010
## 10   0  2381741              Algeria 414578   2   AG 2010
```

That looks good, the only thing is, I can tell you that all the data in this file is fron 2010, so we do not really need the last column. In order to skip it during import, we can use the `colClasses` argument, and setting the seventh argunemt to `NULL`

```
# import full table except for 7th column (year)
population2010 <- read.csv("data/pop2010.csv",
                           colClasses = c("integer",   # age
                                          "integer",   # area
                                          "character", # name
                                          "integer",   # population
                                          "integer",   # sex
                                          "character", # country id (FIPS)
                                          "NULL"))     # year - skip
# check how it looks
head(population2010)
```

```
##   AGE AREA_KM2                 NAME   POP SEX FIPS
## 1   0      180                Aruba   660   1   AA
## 2   0      180                Aruba   653   2   AA
## 3   0      443  Antigua and Barbuda   720   1   AC
## 4   0      443  Antigua and Barbuda   688   2   AC
## 5   0    83600 United Arab Emirates 40770   1   AE
## 6   0    83600 United Arab Emirates 38987   2   AE
```

```
tail(population2010)
```

```
##       AGE AREA_KM2     NAME POP SEX FIPS
## 69079  95   386847 Zimbabwe 506   0   ZI
## 69080  96   386847 Zimbabwe 340   0   ZI
## 69081  97   386847 Zimbabwe 223   0   ZI
## 69082  98   386847 Zimbabwe 142   0   ZI
## 69083  99   386847 Zimbabwe  88   0   ZI
## 69084 100   386847 Zimbabwe 116   0   ZI
```

Comma separated values (`.csv`) is one of the preferred formats to import data from, but R allows you to import from a variety of other formats, although this can sometimes get a bit more messy. This time we will also first download the file, before importing a table from one of the spreadsheets. This is from https://data.gov.uk/dataset/social_trends, part of the government's open data access initiative and a great resource!

```
## 1.2  Download and import an Excel file
################################################################################
# url of the .xls file we want (using paste only to keep code under 80 chars:)
data.url <- paste("http://www.ons.gov.uk/ons/rel/social-trends-rd/social-",
            "trends/social-trends-41/income-and-wealth-data.xls", sep="")

# download location
data.location <-  paste( "data", "income-and-wealth-data.xls", sep = "/")

# download - Excel files are binary, so set the mode to "wb"!!
download.file(data.url, data.location, mode="wb")
```

You can now have a look in the data folder to check the file has been correctly downloaded and inspect it in Excel. We will import the table from the third worksheet, named "Table 1", on peopple's perceptions of the current economic situation. Close the Excel file before proceding! Several solutions are available for importing Excel files into R, a nice overview can be found http://www.r-bloggers.com/read-excel-files-from-r/. In this practical we will use the `xlsx` package:

```
## Importing the data from an .xls file
require(xlsx)
# if you get an error telling you there is no package called 'xlsx' run:
# install.packages("xlsx")

# let's see what happens if we import the whole sheet
economic.situation <- read.xlsx(data.location, sheetIndex = 3)
```

Have a look at `economic.situation`. [1] It is not ideal, empty rows and columns are imported, as is the text at the top and the bottom of the worksheet. Luckily, `read.xlsx` has plenty of arguments that allow us to specify more precisely what we want to import. In this case, we can go one step further, and note that there are actually three separate tables in this worksheet, so it might be easiest to import them separately:

```
## using rowIndex and colIndex select each subtable individually:
world.situation <- read.xlsx(data.location, sheetIndex = 3,
                         rowIndex=c(4, 6:8), colIndex = c(1:7))

UK.situation <- read.xlsx(data.location, sheetIndex = 3,
                            rowIndex=c(4, 11:13), colIndex = c(1:7))

household.situation <- read.xlsx(data.location, sheetIndex = 3,
                          rowIndex=c(4, 16:18), colIndex = c(1:7))
```

Finally, sometimes we need to extract the data from a zipped file, this can also be done directly form R [2]. And to try out another format we will import an SPSS file as well.[3]

```
## 1.3  Download, unzip and import a .dat file
################################################################################
# url of the .zip file we want
data.zip.url <- paste("http://spss.allenandunwin.com.s3-website-ap-southeast-2.",
                      "amazonaws.com/Files/survey.zip", sep="")
```

[1] Are you getting an error? That may be because you still have the Excel file open!

[2] This should work even if no winzip utility is installed on the machine?

[3] The data file is supplementary material to the SPSS Survival Manual from a survey designed to explore the factors that impact on respondents' psychological adjustment and wellbeing.

```
temp <- tempfile()
download.file(data.zip.url, temp)
# check what is in the zip file using list (doens't extract anything)
unzip(temp, list=TRUE)
```

```
##         Name Length                Date
## 1 survey.sav  84640 2015-12-22 12:09:00
```

```
# Only one file, that's the one we want to extract to the data folder
unzip(temp, "survey.sav", exdir = "data")
unlink(temp)
```

You can now check the `data` folder and you should find the `survery.sav` file there. Even if you don't have SPSS installed on your computer, you can now open it using R and the `foreign` package:

```
require(foreign)
# if you get an error telling you there is no package called 'xlsx' run:
# install.packages("foreign")

# import the data as a data frame:
data.location <- paste("data","survey.sav", sep="/")
ed.psy.survey <- read.spss(data.location,  to.data.frame=TRUE)
# check what it looks like
ed.psy.survey[1:5,1:5]
```

```
##    id     sex age             marital child
## 1 415 FEMALES  24  MARRIED FIRST TIME   YES
## 2   9   MALES  39 LIVING WITH PARTNER   YES
## 3 425 FEMALES  48  MARRIED FIRST TIME   YES
## 4 307   MALES  41           REMARRIED   YES
## 5 440   MALES  23              SINGLE    NO
```

Now we have the data, before we continue we'll just do a bit of housekeeping and clear our workspace of the objects we don't need anymore (including the survey dataset, which we will not use in this practical)

```
# CLEAN UP!
rm(economic.situation, ed.psy.survey, data.location, data.url, data.zip.url, temp)
```

4.3.2 Data Tidying

The population2010 data.frame is already pretty tidy! The only issue with it is the `SEX` variable, which is coded for men (SEX==1), women (SEX==2), and both (SEX==0). We really only need to remove the rows with the values for (SEX==0), but we can use this opportunity to perform a data check as well, while practicing the `spread()` and `gather()` functions:

First let's try out our technique on a small subset of the data - this is good practice in general, especially if you are dealing with large datasets. We'll select only the observations for Aruba, and have a look at them:

```
# try out our technique on a smaller subset of the data
test.data <- population2010[population2010$FIPS == "AA", ]
head(test.data)
```

```
##     AGE AREA_KM2  NAME POP SEX FIPS
## 1     0      180 Aruba 660   1   AA
## 2     0      180 Aruba 653   2   AA
## 457   1      180 Aruba 651   1   AA
## 458   1      180 Aruba 645   2   AA
## 913   2      180 Aruba 643   1   AA
## 914   2      180 Aruba 636   2   AA
```

We want to reshape the table so that the values of `SEX` will become new column names (i.e. *keys*), and that the *values* for these new keys will be the values from the variable `POP`. This means the `spread()` functions should look like this:

```
tidy.test <- spread(test.data, SEX, POP )
head(tidy.test)
```

```
##   AGE AREA_KM2  NAME FIPS    0   1   2
## 1   0      180 Aruba   AA 1313 660 653
## 2   1      180 Aruba   AA 1296 651 645
## 3   2      180 Aruba   AA 1279 643 636
## 4   3      180 Aruba   AA 1266 634 632
## 5   4      180 Aruba   AA 1251 627 624
## 6   5      180 Aruba   AA 1243 624 619
```

```
# and for clarity, let's rename the columns:
colnames(tidy.test)[5:7] <- c("both", "male", "female")
```

We can now check if the totals for men and women actually match, before we discard the column with the sum of both:

```
# calculate sum of males and females
tidy.test$check <- tidy.test$male + tidy.test$female

# compare it with the values already in the table:
all.equal(tidy.test$both, tidy.test$check)
```

```
## [1] TRUE
```

```
# looks good, now we can remove both total columns:
tidy.test$check <- NULL
tidy.test$both <- NULL
```

And finally we have to use `gather()` to get back to a tidy table. Remember, with gather you need to pass the *names* of the new variables that are now the *key* and the *value*, and the column names which hold them:

```
tidy.test <- gather(tidy.test, sex, population, 5:6)
```

If you are happy with the test run, you can now try it on the whole `population2010` table.

Most of the time you will not be lucky enough to work with as nicely formed datasets as the population one. But the same tools can be used to disentangle much more messy tables, such as theo ones we extracted from the Excel file above.

Let's have a look at one of the three files, e.g. `household.situation`, and see how it could be tidied up. What are the variables (that should be in the columns), and what are the observations (that should have one row each)?

```
# first let's remane the column names
colnames(household.situation) <- c("perception", "inc.le.20",
                                   "inc.20.to.39", "inc.40.to.59",
                                   "inc.60.to.99", "inc.gt.100", "all")
household.situation
```

```
##               perception inc.le.20 inc.20.to.39 inc.40.to.59 inc.60.to.99
## 1      Good or very good        28           44           54           64
## 2    Neither good or bad        47           42           35           36
## 3        Bad or very bad        25           14           11            1
##   inc.gt.100 all
## 1         63  40
## 2         34  43
## 3          3  18
```

In fact the whole table needs to be transposed, so that each population group represents one observation, and the proportion answering each question are the variables. In order to do that we need to first gather the data in long form, before spreading it out again wide.[4]

```
X.household.situation <- gather(household.situation, income.group, proportion, 2:7)
tidy.household.situation<- spread(X.household.situation, perception, proportion)
rm(X.household.situation)
```

Don't forget, we have two more tables, one for each perception question. If we want to merge them together at the end, we need to be clear that each observation refers to one of the questions:

```
tidy.household.situation$perception <- "HH"
```

Now you can repeat the tidying for the UK and world perceptions, and once all three tables are tidy, you can merge them together using `rbind()`:

```
tidy.world.situation$perception <- "W"
# merge all the tables together
tidy.economic.situation <- rbind(tidy.household.situation,
                                 tidy.UK.situation,
                                 tidy.world.situation)
```

Finally, you can now clear your workspace using `rm()` as we did before, to remove everything except for `tidy.population2010` and `tidy.economic.situation`.

[4] The `t()` function will transpose a data frame in R, try it out to see if it is a useful alternative to gather and spread.

5 Efficient Coding

5.1 Standard control structures

5.1.1 Conditional execution

5.1.2 Looping

5.1.3 PRACTICAL

5.2 Vecotrisation and `apply` family of funcitons

5.2.1 PRACTICAL

benchmarking apply vs for loops

5.3 Writing your own functions

5.3.1 objects, types, environments

5.3.2 passing arguments

5.3.3 PRACTICAL

5.4 Data manipulation with `dplyr`

5.4.1 Subsetting

- `filter`
- `sample`
- `slice`
- `distinct`
- `select`

5.4.2 Grouping

- `group_by`

5.4.3 Summarizing

- with own function

5.4.4 Making new variables

- `mutate`

5.4.5 Piping/chaining daisies

5.4.6 PRACTICAL

5.5 FINAL PRACTICAL

something along the lines of:

- Fun1: a function to be called in summarize or mutate (e.g. z-score)
- Fun2: a chain (that calls Fun1), and then filters the table in some way e.g. subset for each country
- Fun3: a nice plotting function that takes the result of Fun2 and plots it, using `paste()` for titles etc..

5.6 Accessing Data Using APIs

APIs (Application Programming Interface) allow standardised data access to a variety of web resources. More and more websites are publishing them making it easy for developers and researchers to dynamically access or update content.

> When used in the context of web development, an API is typically defined as a set of Hypertext Transfer Protocol (HTTP) request messages, along with a definition of the structure of response messages, which is usually in an Extensible Markup Language (XML) or JavaScript Object Notation (JSON) format.

[Source: WIkipedia]

With R we can easily handle both processes:

- Input: The HTTP request
- Output: The .json or .xml response

We will use the `httr` package to