R: Core Skills for Reproducible Research

Course Manual with Practical Exercises

Maja Založnik

Contents

1 Introduction **2**

1.1 Blurb . . . 2

1.2 Key Topics . . . 2

1.3 Course information . . . 2

2 Reproducible Research **3**

2.1 Why? . . . 3

3 Set-up **3**

3.1 RStudio . . . 3

3.2 Project management . . . 3

3.3 Literate programming . . . 3

4 Workflow **5**

4.1 Importing data . . . 5

4.2 Data tidying . . . 5

4.3 PRACTICAL: new R project . . . 9

4.4 PRACTICAL: Import and clean some data . . . 10

5 Efficient Coding **17**

5.1 Standard control structures . . . 17

5.2 Vecotrisation and `apply` family of funcitons . . . 20

5.3 Writing your own functions . . . 22

5.4 Data manipulation with `dplyr` . . . 27

5.5 FINAL PRACTICAL . . . 28

5.6 Accessing Data Using APIs . . . 29

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 functions
- 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 research 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 increasingly 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 research 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 coming 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 access 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

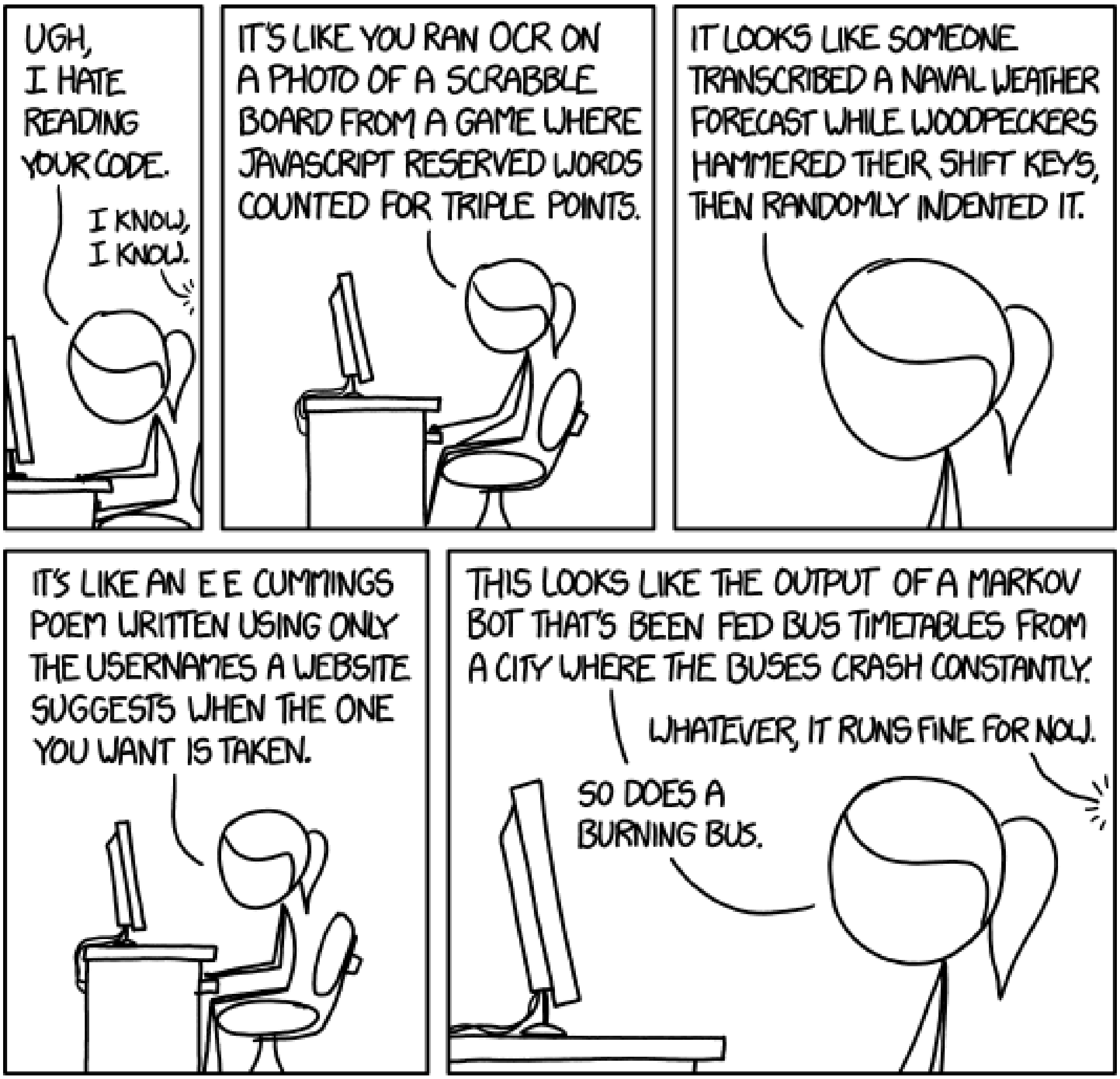


Figure 1: Code Quality part 2. (https://xkcd.com/1695/)

3.3.3 Commenting

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 crucial for the integrity 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 input 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 functions. 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 functions 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 syntax 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 syntax 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()

Separate and unite are straightforward helper functions 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
```

For an excellent write-up of the main `tidyr` functions see Garrett Grolemund's post here http://garrettgman.github.io/tidying/.

For a quick tidyr cheat-sheet stick this to your wall: Data Wrangling Cheatsheet

4.3 PRACTICAL: new R project

The complete documentation for this course is available as an RStudio project in a public github repository: https://github.com/majazaloznik/RepResCoreSkillsR or for extra convenience: http://tinyurl.com/RCSRepRes.

In this first part of the practical you will set up a new RStudio project mirroring the structure of the github repository for this course.

Unless you have brought your own laptop, you will be completing this project on the local drive of the computers here. This unfortunately means you will have to transfer your work via USB, email or other means in order to keep it for your record. The complete materials will however remain available to you on-line at the above addresses.

4.3.1 Create new RStudio project

1. Open RStudio
2. Select the project menu in the top right-hand corner and select `New Project`
3. Select New Directory and choose the name of your project (e.g. RRepResCourse)[1].

4. RStudio has now created a new project file in your folder, which you can see in the Files pane.
5. Run the command `getwd()` in the console. RStudio automatically sets the working directory in the top level of your project folder.
6. Click on the project menu again and select Project Options. On the options "Restore .RData to the workspace at start-up" and "Save workspace to .RData on exit" select No. In fact, you should set that as a global option (Tools/Global Options) – for truly reproducible research you should never have to load a previous workspace!

4.3.2 Folder and File Structure

1. Using the New Folder button in the Files pane, create three folders called (some equivalent of)

- data
- scripts
- figures

[1] As a general rule you should avoid spaces in file and folder names, although you will probably be fine if you ignore this advice.

- presentations

2. For the rest of this practical, all you need is to download the file `pop2010.csv` from the github repo into your `data` folder.
3. You can also download the manual and slides to your `presentation` folder.
4. Create an .RProfile file using the command `file.edit(".Rprofile")`. This will open a new script tab for you to edit. The content of the .RProfile gets automatically run every time you open your project. It is therefore a good idea to e.g. load any packages you will need from your .RProfile, instead of doing it manualy every time.

For this practical you will need the following packages (type this into your .RProfile file and save)

```
require(xlsx)
require(forein)
require(dplyr)
require(tidyr)
```

4.4 PRACTICAL: Import and clean some data

4.4.1 Downloading and importing data

First create a new `.R` script file in your `scripts` folder or equivalent via the drop-down menu or using `Ctrl+Shift+N`. 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()` function 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"
```

```r
# 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 from 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 argument to `NULL`

```r
# 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
```

```r
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 people's perceptions of the current economic situation. Close the Excel file before proceeding! 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`. [2] 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 [3]. And to try out another format we will import an SPSS file as well.[4]

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

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

[4]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 well-being.

```r
## 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="")
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
```

```r
# 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:

```r
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 any more (including the survey dataset, which we will not use in this practical)

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

4.4.2 Data Tidying

In the second part of this practical we will use the functions from the `tidyr` package to tidy up the two datasets.

```r
## 2.1 tidy up the population data
################################################################################
require(tidyr)
```

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 practising 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
```

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

# so now we have:
head(tidy.test)
```

```
##   AGE AREA_KM2  NAME FIPS male female
## 1   0      180 Aruba   AA  660    653
## 2   1      180 Aruba   AA  651    645
## 3   2      180 Aruba   AA  643    636
## 4   3      180 Aruba   AA  634    632
## 5   4      180 Aruba   AA  627    624
## 6   5      180 Aruba   AA  624    619
```

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:

```r
tidy.test <- gather(tidy.test, sex, population, 5:6)
# and let's check it again:
head(tidy.test)
```

```
##   AGE AREA_KM2  NAME FIPS  sex population
## 1   0      180 Aruba   AA male        660
## 2   1      180 Aruba   AA male        651
## 3   2      180 Aruba   AA male        643
## 4   3      180 Aruba   AA male        634
## 5   4      180 Aruba   AA male        627
## 6   5      180 Aruba   AA male        624
```

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

```
##   AGE AREA_KM2             NAME FIPS both male female
## 1   0        2           Monaco   MN  216  110    106
## 2   0        7        Gibraltar   GI  405  209    196
## 3   0       21            Nauru   NR  253  115    138
## 4   0       21 Saint Barthelemy   TB   80   41     39
## 5   0       26           Tuvalu   TV  233  119    114
## 6   0       28            Macau   MC 5052 2586   2466
```

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

```r
## 2.2 tidy up the perception data
###############################################################################
```

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 the 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)?

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

```
##               perception inc.lt.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.[5]

```r
# transpose using gather and spread
X.household.situation <- gather(household.situation, income.group, proportion, 2:7)
tidy.household.situation <- spread(X.household.situation, perception, proportion)
# let's also rename the column names in keeping with the convention of avoiding spaces
colnames(tidy.household.situation) <- c("income.group", "bad", "good", "neutral")
# check the result and remove the temporary table
tidy.household.situation
```

```
##   income.group bad good neutral
## 1    inc.lt.20  25   28      47
## 2 inc.20.to.39  14   44      42
## 3 inc.40.to.59  11   54      35
## 4 inc.60.to.99   1   64      36
## 5   inc.gt.100   3   63      34
## 6          all  18   40      43
```

```r
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:

```r
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()`:

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

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

[5] 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

This section covers some of the most important skills to improve the efficiency, readability, and reproducibility of your R code. The standard control of the flow of your code that can be achieved with `ifelse` statements and looping is covered briefly, however you are encouraged in particular to explore the advantages of *vectorised* R code. Writing your own functions will greatly streamline your work, as well as forcing you to think in more abstract terms about your analysis - making it easily transferable and reproducible as opposed to limited to the specific situation you find yourself analysing at the moment.

Finally, the relatively new `dplyr` family of functions is one of the most powerful recent developments in the R coding world. While all of the data processing capabilities of `dplyr` existed in R before, `dplyr` brings them together in a comprehensive and systematic way, that allows for cleaner code that is easier to read, and faster to run. It also integrates logically with the `tidyr` family of functions described earlier, but it's most exciting and revolutionary aspect is it's assimilation of the *piping* or *chaining* of successive data processing functions - originally developed in the `magrittr` package, and now rightly becoming mainstream R practice.

5.1 Standard control structures

An indispensable gain in efficiency of your programming can be achieved by using *control strucures* to control the execution of your code. These can be divided into *conditional execution* structures (`if` and `else` type functions) and *looping* structures. However, as we shall see in the next section, there are some very good reasons to avoid looping in R!

5.1.1 Conditional execution

The standard syntax for conditional execution is as follows:

```
if (condition) {
  # do something
} else {
  # do something else
}
```

In fact, you may also use only the `if()` construct on it's own:

```
if (condition) {
  # do something
}
```

The if/else syntax also works in a single line, where you can dispense with the curly braces:

```
if (x >= 0)  print("Poz") else print("Neg")
```

While this is more compact, it can impact readability, and can also make your code more difficult to debug and extend. Using curly braces and indenting the code properly will make it clearer to the reader, and also easier to e.g. extend via nesting:

```
x <- runif(1) # randum number from uniform distribution [0,1]
if (x >= 0.6) {
  print("Good")
} else {
```

```
  if (x <= 0.4) {
    print("Bad")
  } else {
    print("Not Sure")}
}
```

The conditions to be evaluated are:

```
x == y   # x is equal to y
x != y   # x is not equal to y
x > y    # x is greater than y
x < y    # x is less than y
x <= y   # x is less than or equal to y
x >= y   # x is greater than or equal to y
x %in% y # x is located in y
TRUE     #
FALSE    #
```

And these can further be combined using standard logical operators:

```
! x       # NOT
x & y     # AND
x | y     # OR
xor(x, y) # exclusive OR
```

What if you want to run a conditional statement over an entire vector? You might be tempted to jump to the next section on looping, and construct a loop going over each element of the vector and evaluating the condition. This would of course work, but it would be a very inefficient way of coding, and would not be taking advantage of the efficiencies of vectorisation in R (covered in the next subsection). In such cases, you should use the *vectorised* form of the if/else construct:

```
ifelse(condition, yes, no)
```

Where `yes` is the value to be returned if the condition is satisfied, and `no` if not. Similarly as above, `ifelse()` statements can also be nested.

```
x <- runif(20) # 20 randum numbers from uniform distribution [0,1]
ifelse(x >= 0.6, "G",
       ifelse(x<=0.4, "B", "N"))
```

```
##  [1] "B" "G" "G" "B" "B" "G" "G" "N" "B" "G" "G" "N" "B" "G" "G" "B" "N"
## [18] "B" "G" "B"
```

5.1.2 Looping

R distinguishes two types of loops: - ones that execute a function a predetermined number of times, as determined by an *index* [`i`] - ones that execute a function until a condition is met

The `for()` loop construct takes the following form:

```
for (i in seq) expr
```

Again, using curly braces is usually preferred, for loops can be nested and the indices need not be integers:

```
mat <- matrix(NA, nrow=3, ncol=3)
for (i in 1:3){
  for (j in 1:3){
    mat[i,j] <- paste(i, j, sep="-")
  }
}
mat
```

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

While loops take the following form:

```
while(cond) expr
```

```
cumsum <- 0
while(cumsum <= 3) {
  cumsum <- cumsum + runif(1)
  print(cumsum)
}
```

```
## [1] 0.8008
## [1] 1.364
## [1] 1.569
## [1] 2.482
## [1] 2.965
## [1] 3.948
```

A repeat loop is similar, but we must explicitly add a `break` to specify when to exit the loop:

```
cumsum <- 0
repeat {
  cumsum <- cumsum + runif(1)
  print(cumsum)
  if (cumsum >= 3) break
}
```

```
## [1] 0.73
## [1] 1.639
## [1] 1.82
## [1] 2.412
## [1] 3.311
```

Both of these constructs should be used with great care, as careless specification of the exiting condition can leave you stuck in an infinite loop. Try running the last example without the line specifying the break! Luckily RStudio allows you to interrupt such an endless loop using the little red stop button in the top right corner of the console window.

5.2 Vecotrisation and `apply` family of funcitons

Looping functions - the `for()` loop in particular - are very intuitive and mastering them can represent a quick capability boost for a new R programmer. It is however highly recommended that you spend some time mastering the related `apply` family of functions, which should cover most of your looping needs. The genera rule is this: If you need to apply an expression over a series of elements and the order in which you do this is important, then use a loop. If the order is not important, take advantage of `apply`. In many circumstances this can improve the speed of your code, but in all cases it will make your code simpler and easier to read.

The underlying logic of the `apply` family is starting out with some data structure (a vector, matrix, data.frame etc.), we want to split it into constituent parts, apply a function on each of them, and combine them back[6]. We might for example want to apply a function on every row of a data.frame, every element of a vector, or every column in a matrix.

5.2.1 `apply()`

The `apply()` function will apply a function to either the rows or the columns of a matrix. It's basic structure is:

```
apply(X, MARGIN, FUN, ...)
```

Where X is a matrix (if it is a data.frame, R will coerce it to a matrix), `MARGIN == 1` indicates rows, and `MARGIN == 2` indicates columns. A simple example of its use is to calculate row and column totals:

```
mat <- matrix(1:9, 3,3)
# row totals
apply(mat, 1, sum)
```

```
## [1] 12 15 18
```

```
# column totals
apply(mat, 2, sum)
```

```
## [1]  6 15 24
```

In passing the function `FUN` in the example here we used built in function `sum`, but the real power of `apply` comes from integrating it with user defined functions. These are covered in the next section, but here is a quick example of how an in-line function can be used to find the second largest value in each row of a matrix.

```
mat <- matrix(sample(1:100, 25), 5,5)
mat
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]   92   71   86   33   40
## [2,]   37   15   99   56   39
## [3,]    3   49   61    8   88
## [4,]   52   47   89   21   69
## [5,]   11   95   74   59    4
```

[6] This idea comes from Hadley Wickham's paper on the split-apply-combine strategy of data analysis: http://vita.had.co.nz/papers/plyr.html

```r
# find the second largest value in each row
apply(mat, 1, function(x) sort(x, decreasing = TRUE)[2])
```

```
## [1] 86 56 61 69 74
```

```r
# and for comparison, here is how we would do this using a for loop
out <- vector()
for (i in 1:nrow(mat)) {
  out[i] <- sort(mat[i,], decreasing = TRUE)[2]
}
out
```

```
## [1] 86 56 61 69 74
```

5.2.2 lapply() and sapply()

The functions `lapply()` and `sapply()` both apply a function to a vector, and the first returns a list back, while the second will try to simplify and return a vector.

It is important to note that in R there are two types of vectors: i) atomic vectors and ii) lists.

Furthermore, data frames in R are also represented as lists, with each column is an element of the list, represented by a vector.

So this means both these functions can be applied to atomic vectors, to data frames, or to other types of lists:

```r
# a list of elements with different lengths:
test <- list(a = 1:5, b = 20:100, c = 17234)
lapply(test, min)
```

```
## $a
## [1] 1
##
## $b
## [1] 20
##
## $c
## [1] 17234
```

```r
sapply(test, min)
```

```
##     a     b     c
##     1    20 17234
```

```r
# a data frame (list of three vectors of equal length):
test <- data.frame(a = 1:5, b = 6:10, c = 11:15)
lapply(test, mean)
```

```
## $a
## [1] 3
##
```

```
## $b
## [1] 8
##
## $c
## [1] 13
```

```
sapply(test, mean)
```

```
##  a  b  c
##  3  8 13
```

```
# an atomic vector (this is rather silly, since sqrt(X) would work the same)
# but is added for completeness
test <- 1:3
lapply(test, sqrt)
```

```
## [[1]]
## [1] 1
##
## [[2]]
## [1] 1.414
##
## [[3]]
## [1] 1.732
```

```
sapply(test, sqrt)
```

```
## [1] 1.000 1.414 1.732
```

By writing more elaborate functions and passing them as the argument to any of the `apply` family of functions, this seemingly simple construct can become incredibly powerful - as well as making the code eminently readable.

5.3 Writing your own functions

One of the greatest strengths of R comes from writing your own functions. This not only allows you to repeat the same procedure consistently, but makes your code more structured and readable reduces chance of error, and will further strenghtens your reproducibility mentality.

The basic construct is as follows:

```
function.name <- function(arguments, ...) {
  expression
  (return value)
}
```

we have already seen the in-line version of the function call in the apply example above, remember:

```
function(x) sort(x, decreasing = TRUE)[2]
```

Here our function takes a single argument (`x`), evaluates the expression (`sort(x, decreasing = TRUE)[2]`), and returns the value of that expression. This only works if the function has only a single expression, in which case the evaluated expression is returned. Otherwise we have to explicilty state what we want returned. We can rewrite this function in the more elaborate mode:

```
FunSecondLargest <-function(x) {
  r <- sort(x, decreasing = TRUE)[2]
  return(r)
}
# now let's try it out with a sample vector
test.vector <- tidy.population2010$population

FunSecondLargest(test.vector)
```

```
## [1] 14642884
```

We can also now use this function directly in the apply call we used before:

```
apply(mat, 1, FunSecondLargest)
```

```
## [1] 86 56 61 69 74
```

We can also quickly rewrite the funciton to instead find the n-th largest value in the vector, by adding an additional argument `n`. And don't forget to write sensible comentary about what you are doing - at least for the benefit of your future self!

```
# Function for extracting the n-th largest value from a vector
# Agruments:
#   x - vector
#   n - optional integer value for rank
# Output:
# Returns single value
FunNthLargest <-function(x, n=1) {
  r <- sort(x, decreasing = TRUE)[n]
  return(r)
}
# by default n=1, so it will find the largest value if we don't specify
FunNthLargest(test.vector)
```

```
## [1] 15120232
```

```
FunNthLargest(test.vector, n=2)
```

```
## [1] 14642884
```

```
FunNthLargest(test.vector, n=3)
```

```
## [1] 13601669
```

We could e.g. further generalise this function to look for the n-th smallest value, by adding another argument for the `TRUE/FALSE` value that gets passed to `decreasing` etc. Note unlike the argument `x`, the argument `n` has a default value (=1). This means we do not have to explicitly specify it unless we want it to be a different value.

Functions have their own local environment, which is not accessible from the global environment. This means that whatever calculations are evaluated inside the function call do not clutter your workspace, but also their results are not accessible unles you explicilty return them from the function. Thus the object `r` will not be found in the global environment, instead we will get the error:

```
r
Error: object 'r' not found
```

We can also have our funciton return several outputs for example:

```
FunNthLargestElaborate <-function(x, n=1) {
  r <- sort(x, decreasing = TRUE)[n]
  desc <- paste("Rank", n, sep=":")
  return(c(desc, r))
}
FunNthLargestElaborate(test.vector, 3)
```

```
## [1] "Rank:3"   "13601669"
```

As we have seen before with if/else statements and loops, functions can also be nested – as well as combined with if/else statements and loops! It is good practice to try to keep your code modular: keep your functions short and call them from eachother. This again makes it easier for the reader to understand what is going on, and easier for you to find errors or update your code.

From a project management point of view it is also good practice to store all your functions in a separate file, which you `source()` at the begining of each session. You can even add `source("00-MyFunctions.R")` to your `.RProfile` file, which means all your bespoke functions will be automatically uploaded at the start of each session.

5.3.1 PRACTICAL

Practice conditional expressions and logical operators by seeing if you can figure out the results of the following expressions, then check them in R:

```
## 3.1 Practice conditional expressions and logical operators
################################################################################
x <- 1:7
y <- -3:3
x
```

```
## [1] 1 2 3 4 5 6 7
```

```
y
```

```
## [1] -3 -2 -1  0  1  2  3
```

```r
# try the following:
(x == y)
(x > abs(y))
(x > 3) & (x < 5)
(x > 3) | (x < 5)
xor((x > 3), (x < 5))
(-1 %in% y)
(3 %in% y) & (3 %in% x)
(3 %in% y) & !(3 %in% x)
```

Use a `for()` loop to go through every row of `tidy.economic.situation` (tip: `nrow()` will tell you how many iterations you need): * for each row add up the proportions for all three answers (columns two to four) * use an `if/else` construct to check if the total equals 100 + if it does, use `print` to print out an OK message + if it doesn't, print out a different message, one created using `paste` - so you can include the information on *which* row you have found the error.

```r
## 3.2 Check consistency of tidy.economic.situation proportions ising IF - ELSE
###############################################################################
for (i in 1: nrow(tidy.economic.situation)) {
  if(???){
    ???} else {
      ???}
}
```

Now write a function for the row checking you just did inside the `for()` loop. This is simply generalising the if/else expression to take a supplied argument instead of explicitly naming the row: * Make sure you *document* your function correctly! * the input for the function should be a row * the output of the function should be a variable called `message` - "OK" or "Not OK" * Use the framework below:

```r
## 3.3 Write a function to check consistency of tidy.economic.situation proportions
###############################################################################
FunRowCheck <- function(x) {
  message <- if(???){
  } else {
  }
  return(message)
}
```

Now test out your function on a single row:

```r
FunRowCheck(tidy.economic.situation[1,2:4])
```

```
## [1] "OK"
```

If it works correctly, you can now try using your new function inside an `apply` construct.

Remember, `apply` will evaluate the expression along the *whole* row, and you want to apply it only to columns 2:4, so make sure you don't pass the whole table to `apply`. When you are happy with the result, append it to the table as an additional column:

```
tidy.economic.situation$test <- apply(???)
```

Your table should now look like this:

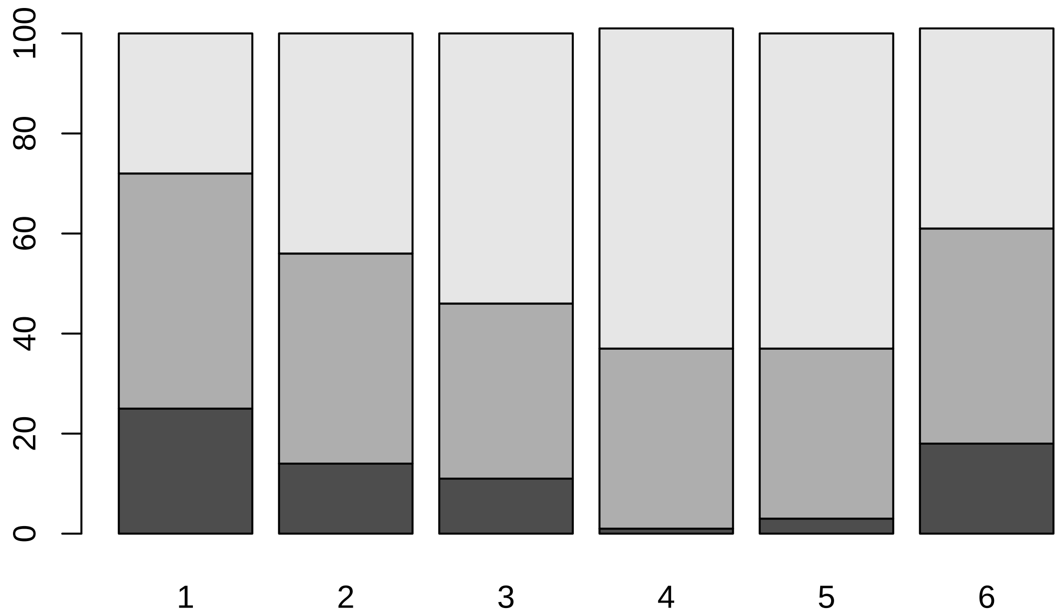
```
tidy.economic.situation
```

```
##    income.group bad good neutral perception   test
## 1     inc.lt.20  25   28      47         HH     OK
## 2  inc.20.to.39  14   44      42         HH     OK
## 3  inc.40.to.59  11   54      35         HH     OK
## 4  inc.60.to.99   1   64      36         HH Not OK
## 5    inc.gt.100   3   63      34         HH     OK
## 6           all  18   40      43         HH Not OK
## 7     inc.le.20  76    8      17         UK Not OK
## 8  inc.20.to.39  80    5      15         UK     OK
## 9  inc.40.to.59  83    4      13         UK     OK
## 10 inc.60.to.99  92    4       4         UK     OK
## 11   inc.gt.100  89    0      11         UK     OK
## 12          all  80    6      15         UK Not OK
## 13    inc.le.20  77    6      17          W     OK
## 14 inc.20.to.39  79    3      18          W     OK
## 15 inc.40.to.59  85    2      12          W Not OK
## 16 inc.60.to.99  91    1       8          W     OK
## 17   inc.gt.100  92    0       8          W     OK
## 18          all  80    4      16          W     OK
```

If it doesn't have a look at the `02-FunctionsAndLoops.R` file in the scripts folder for the solution. If it does and you have plenty of time, you can try additionally: * writing another function that will simply sum the three columns * applying it to the table and appending the new variable (`total`) * creating three new variables that are correctly scaled to sum up to 100 (e.g. `bad.scaled`, `good.sscaled`, `neutral.scaled`).

To finish off this practical session we will write a function for plotting our table, using the `barplot()` function. This function only accepts vectors or matrices, but because our data is in a data.frame, we need to use `as.matrix()` for it to work, in addition to `t()` for transposing it. Here is the code for the most stripped down stacked barplot of the people's perceptions of their household financial situation. Note that the order of the columns had to be changed to make the more logical order of bad - neutral - good.

```
barplot(t(as.matrix(tidy.economic.situation[1:6,c(2,4,3)])))
```



Now expand the barplot function - use the help documentation in the help tab:

- add names to the x-axis
- add legend text
- add a main title
- feel free to explore additional arguments to the plot!

Once you are happy with your plot, enclose it in a function, so that you can pass it each of the three subsets of the table individually, and the function will additionally also change the plot's title to the correct one.

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

Figure 2: Photo by Joe Cross - https://www.flickr.com/photos/jaycross/2869212451

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