7316 - Introduction to R

Module 2: Data manipulation in R

Teacher: Mickaël Buffart (mickael.buffart@hhs.se)

Table of Contents

This document was generated with R markdown.

1 Packages in R

Packages in R are kinds of extensions. They can bring new functions or new data to the base R software, similar to user-written commands (think `ssc install`) in Stata, libraries in Python (think `pip install`), or plugins in Excel. Yet, with Stata or Excel, **most** of the things you do probably use the core Stata commands. In R, most of your analyses will probably be done using packages. Once loaded in the environment, a package behave exactly as a core component of R.

There are two main sources of packages in R:

1. The most important is the CRAN repository. This is the official source of package containing a very extensive list (more than 19'000) with their documentation. To be available on CRAN, packages need to fulfill some quality criteria, checked by a team of volunteers. It does not guarantee complete security, but at least the packages have been reviewed by someone before being available in the repository[1].
2. You can also to install packages directly from github. While github may contain more recent versions of the packages, as well as some that are not available in the official repository, those are **NOT reviewed by anyone**. There is therefore no guarantee at all that the package respects minimum quality criteria, or even that the package works. It might however be useful to use in some cases.

1.1 Installing packages from the official repository (CRAN)

- To install a package from RStudio, you can click on **Tools** > **Install packages…**, type the name of the package you would like to install, and click **Install**.
- Alternatively, you can use the function (preferably in the console) `install.packages()`
- To begin with, let's install two packages:
  - `tidyverse`, developed by Hadley Wickham: "The tidyverse is an opinionated collection of R packages designed for data science. All packages share an underlying design philosophy, grammar, and data structures"[2].
  - `remotes`, developed by Gábor Csárdi et al.: `remotes` allows your to download and install packages from other sources than the official repository, including github.
  - `rio`, developed by Jason Becker et al.: `rio` is a package for easy data import, export (saving), and conversion.

```
install.packages("tidyverse")
install.packages("remotes")
install.packages("rio")
```

1.2 Installing packages from Github

To install package from github, you can use the following command:

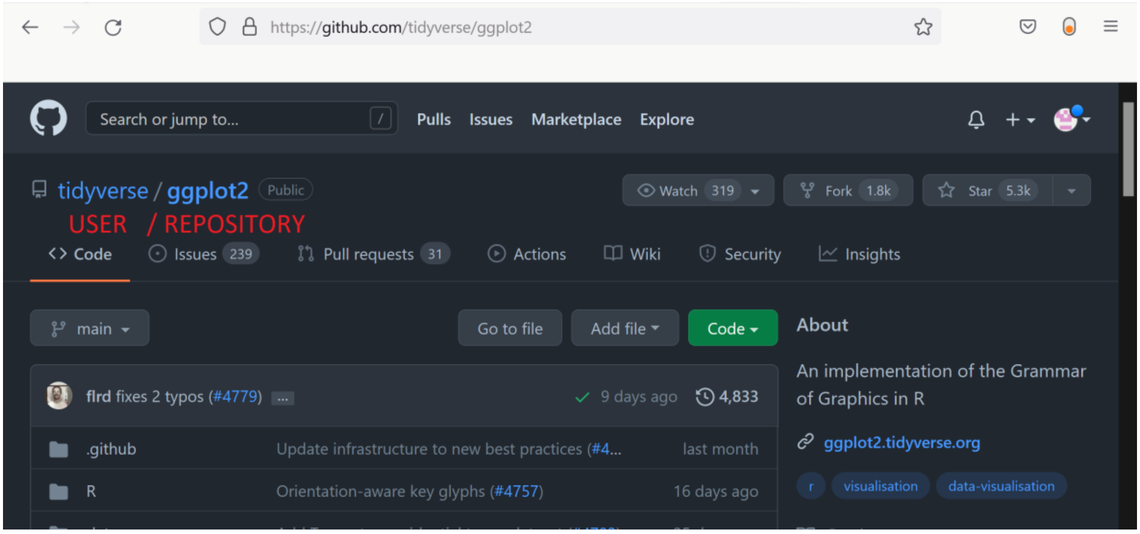
---

[1] The CRAN also provides the list of authors for each packages they publish: you can assess if the author is a famous unknown, or someone from a serious institution.

[2] Source: https://www.tidyverse.org/

```r
remotes::install_github("user/repository")
```

where `user` is the name of the user on github who posted the package, and `repository` is the name of the package on github.



Github package example

1.3 Using functions from a package in your scripts

- The best way to call a function from a package is through the following code: `package_name::function_name()`. With this, each function call is precisely related to the package it is from. This is what we used in the example above to install a package from github.
- In some cases, especially when you are using a specific package a lot in your script, it can be handy to load the package once for all. To do this, use the `library(package_name)` function at the beginning of your script. If you would like to load all the functions of the tidyverse in your environment, use:

```r
library(tidyverse)
```

2 IMPORTING, EXPORTING, SELECTING

2.1 Importing using `rio`

Previously, importing and exporting data in R was a mess, with a lot of different functions for different file formats. Stata `.dta` files alone required two functions: `read.dta` (for Stata 6-12), `read.dta13` (for Stata 13 and later), etc.

The `rio` package simplifies this by reducing all of this to just one function, `import()`, that automatically determines the file format of the file and uses the appropriate function from other packages to load in a file. It is able to load more than 30 different datafile formats, including, csv, Excel, SAS, SPSS, Stata, Matlab, JSON, and others. If a file is not recognize by `rio`, it is always possible to find and use a specific function for this unrecognized file format.

Let's try. On canvas, I provide you with a zipfile named `7316 - Module 2 - data.zip`. You can download the file, unzip it, and place the data files it contains in a `data` folder[3] in your project directory[4].
Now, your `data` folder contains the School questionnaire data file from the PISA survey (2018) that I downloaded from the OECD website. I provide them in two different format, for the example:

- `cy07_msu_sch_qqq.sas7bdat`: the dataset as an SAS data file.
- `CY07_MSU_SCH_QQQ.sav`: the dataset as an SPSS data file.

Use the following command to load the data in your R environment:

```
# Dataset in SAS format
pisa_sas <- rio::import("data/cy07_msu_sch_qqq.sas7bdat")

# Dataset in SPSS format
pisa_spss <- rio::import("data/CY07_MSU_SCH_QQQ.sav")
```

Note: If a dataset you are interested in is part of a package, you can load it by simply calling its name. For example, let's use the `Wages1` dataset stored in the package `Ecdat`.

```
wages <- Ecdat::Wages1
```

2.2 Display the structure of the dataset

After loading your dataset, you may want to see its structure. You can see the structure in the environment tab, by clicking on the small arrow before the `dataframe` name. Alternatively, you can use the function `str()`. It will display the names of the variables within your `dataframe`, their `types`, and a few first observations.

```
str(pisa_sas)
```

2.3 Exporting data

If you want to save an object, *e.g.* pisa_sas, into a new file, you can use:

```
# SOLUTION 1:
saveRDS(pisa_sas, "data/pisa_sas.Rds")

# SOLUTION 2:
rio::export(pisa_sas, "data/pisa_sas.Rds")
```

The function `saveRDS()` is the base function to save R data object into a file. `rio::export()` is the wrapper from the `rio` package. They both lead to the exact same result.

`Rds` is the standard format to save data in R. I advise you to use it, because you are sure that the data are saved exactly as you see them in R (no loss of information). However, if you need to export the data into a file compatible with another statistical software, you can also use `rio::export()` to export the data into whatever format you wish.
Warning: Some format will result in a loss of data, in case some types or encoding of your dataframe are not compatible with the chosen file format.

[3] In practice, you may name the `data` folder as you like, but it is common practice to name it `data`.
[4] **Do not forget** from Module 1: before starting any new data work, create a new R project in a new directory. The directory will contain your data, scripts, and outputs.

```
# Saving the data into an Excel sheet
rio::export(pisa_sas, "data/pisa_sas.xlsx")

# Saving the data for Stata
rio::export(pisa_sas, "data/pisa_sas.dta")
```

3 Data and the tidyverse

In the recent years, Hadley Wickham introduced the Tidyverse: a set of functions to manipulate data in R.

3.1.1 *Principles of tidy data*

Rules for tidy data (from *R for Data Science*):

1. Each variable must have its own column.
2. Each observation must have its own row.
3. Each value must have its own cell.

A motivating principle behind the creation of the tidyverse was the language of programming should really behave like a language. This means that most of the outputs of the tidyverse could be obtained without it, using only R base functions, with a different syntax. The tidyverse aims at making all the data preparation process more homogeneous: data manipulation in the tidyverse is oriented around a few key "verbs" that perform common types of data manipulation.

3.2 A parenthesis about the tibbles

Last class, we covered `dataframe`, the most basic data object class for data sets with a mix of data class. Today, we introduce one final data object: the `tibble`! The tibble is an alternative to the `dataframe` that has been created as a part of the `tidyverse` package. As `dataframe`, `tibble` contain a table of data, *e.g.* variables as vectors in columns, possibly of different types, and rows as values. In most cases, you would not see the difference of using `dataframe` or `tibble`, but you need to know both, because some functions of the `tidyverse` require the use of `tibble`. Additionally, `tibble` differs from `dataframe` when:

- displaying `dataframe`: it will print as much as much output as allowed by the `max.print` option in the R environment. With large data sets, that is one thousand lines. `tibble` by default print the first 10 rows and as many columns as will fit in the window: more readable.
- matching in `dataframe`: when using the `$` method to reference columns of a data frame, partial names will be matched if the reference isn't exact. This might sound good, but the only real reason for there to be a partial match is a typo, in which case the match might be wrong.

3.2.1 *Creating or converting to tibbles*

The syntax for creating tibbles exactly parallels the syntax for data frames:

- `tibble()` creates a tibble from underlying data or vectors (this is equivalent to `data.frame`).
- `as_tibble()` coerces an existing data object into a tibble.

```
df_tibble <- tibble::as_tibble(pisa_sas)
```

```
class(pisa_sas)
class(df_tibble)
```

You can display `df` and `df_tibble` to see the difference:

```
pisa_sas

df_tibble
```

3.3 Another parenthesis about Pipes

Another famous function from the tidyverse is ***pipe***, denoted `%>%`:

- Pipes allow you to combine multiple steps into a single piece of code.
- Specifically, after performing a function in one step, a pipe takes the data generated from the first step and uses it as the data input to a second step. Example:

```
library(dplyr)
##
## Attaching package: 'dplyr'
## The following objects are masked from 'package:stats':
##
##     filter, lag
## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union
tibble::as_tibble(pisa_sas) %>% class()
## [1] "tbl_df"     "tbl"        "data.frame"
```

This is absolutely equivalent to writing, but the code bellow does not require an extra package (`dplyr`), to run:

```
class(tibble::as_tibble(pisa_sas))
## [1] "tbl_df"     "tbl"        "data.frame"
```

In general, I find that the pipes `%>%` makes the code less intuitive and more prone to errors. It also does not save the intermediary steps. Thus, I don't use it. However, it is often seen in books and example codes in R, so you need to know what it means. You want also use it as you like.

4 TIDY DATASETS

4.1 Extract variables from dataframe

Now, you can see the two datafiles loaded in dataframes your environment, with 21'903 observations. But the dataset is very big. If you are interested in specific variables, you can extract them into a new `dataframe`. For example, to extract, `"CNT"`, `"CNTSCHID"`, `"SC016Q01TA"`, `"SC155Q02HA"`, from `pisa_sas`:

With the tidyverse, it is possible to do it using `select()`.

```
library(dplyr)

# Using brackets
df <- pisa_sas[, names(pisa_sas) %in% c("CNT",
                                        "CNTSCHID",
                                        "SC016Q01TA",
                                        "SC155Q02HA")]
```

```
# Using select
df_selection <- pisa_sas %>% select(CNT, CNTSCHID, SC016Q01TA, SC155Q02HA)
```

4.2 Remove dataframe or variables

You created a `dataframe` called `df`, containing the variables you are interested in. You may want to remove the two prior datasets that you loaded, to save memory:

```
rm(pisa_spss, pisa_sas)
```

The command above removes two complete datasets from your environment. If you want to remove a variable in a dataset, for example, `"CNTSCHID"`, use the following command:

```
df[, "CNTSCHID"] <- NULL
```

It is possible to do the same with `select()`, from the tidyverse:

```
# Using select
df_drop <- df %>% select(-CNTSCHID)
```

Note the `-` before the variable name to drop.

Question: Do you see any way to get the same results using the `names()` function? And with the `$` operator?

4.3 Rename columns

We have already seen in module 1 how to do it.

```
# With base R
names(df)[names(df) == "SC155Q02HA"]<- "new_name"
```

4.4 Extract a random sample from a dataframe

`df` is very big. If you would like to work on a subset of data, you can extract a random sample from your dataset with the following code:

```
df <- df[sample(nrow(df), 5000), ]
```

where `5000` is the number of random rows that you want to extract.

4.5 Gather data

If values for a single variable are spread across multiple columns (e.g. income for different years), gather moves this into single "values" column with a "key" column to identify what the different columns differentiated. In short, gather converts a wide dataset into long. Example:

```
library(dplyr)

# We create a dataframe for the example
earnings_panel <- data.frame(person = c("Elsa", "Mickey", "Ariel", "Gaston",
                                        "Jasmine", "Peter", "Alice"),
                             y1999 = c(10, 20, 17, 19, 32, 22, 11),
                             y2000 = c(15, 28, 21, 19, 35, 29, 15))

# Gather data
earnings_panel <- earnings_panel %>%
  tidyr::gather(key = "year", value = "wage", y1999:y2000)
```

You could also do this without the tidyverse, but that would require more lines of code.

4.6 Spreading data

Spread tackles the other major problem, that often times (particularly in longitudinal data) many variables are condensed into just a "key" (or indicator) column and a value column. In short, spread converts a long dataset into wide. Example:

```
earnings_panel %>% tidyr::spread(key = "year", value = "wage")
```

5 CLEANING DATA

The tidyverse contains many functions to help you cleaning the data. In most cases, they are redundant to base R, but sometimes provide reasonable enhancement to the base R. The main functions are:

1. `filter()` subsets the rows of a data frame based on their values.
2. `mutate()` adds new variables that are functions of existing variables.
3. `summarize()` creates a number of summary statistics out of many values.
4. `arrange()` changes the ordering of the rows.

Note: the first argument for each these functions is the data object.

5.1 Filter data

Filtering keeps observations (rows) based on conditions, just like using use subset conditions in the row arguments of a bracketed subset. Example:

```
# Using brackets
wages[(wages$school > 10) & (wages$exper > 10),]

library(dplyr)
# Using filter
wages  %>% filter(school > 10, exper > 10)
```

Notice a couple of things about the output:

1. It doesn't look like we told `filter()` what data set we would be filtering: that's because the data set has already been supplied by the pipe. We could have also written the filter as:

```
filter(wages, school> 10, exper > 10)
```

2. We didn't need to use the logical `&`. Though multiple conditions can still be written in this way with `filter()`, the default is just to separate them with a comma.

5.2 Mutate data

Creating new variables that are functions of existing variables in a data set can be done with `mutate()`.

mutate() takes as its first argument the data set to be used and the equation for the new variable:

```
# With base R:
wages$expsq <- wages$exper^2

# With the tidyverse:
wages <- wages %>% mutate(expsq = exper^2)
```

5.3 Summarize data

Summary statistics can also be created using the tidyverse function `summarize()`

The `summarize` functions uses summary statistic functions in R to create a new summary tibble, with syntax largely identical to mutate().

Let's try summarizing with the mean() summary statistic.

```
# With the tidyverse:
summary_stat <- wages %>% summarize(avg_wage = mean(wage),
                                    median_wage = median(wage))

# With base R:
summary_stat <- data.frame(avg_wage    = mean(wages$wage),
                           median_wage = median(wages$wage))
```

5.3.1 Summary Statistics functions in R

There are a number of summary statistics available in R, which can be used either with the summarize() command or outside of it:

- `mean()`,
- `median()`,
- `sd()`,
- `cor()`,
- `quantile()`,
- `IQR()`

Warning: If your data contains missing values, the function will send `NA`. To avoid it, use the `na.rm` argument:

```
# No na.rm
mean(df$SC016Q01TA)
## [1] NA
# With na.rm
mean(df$SC016Q01TA, na.rm = TRUE)
## [1] 82.31528
```

- `length()`: gives you the length of a vector, equivalent to `dplyr::n()` in the tidyverse.
  - `length(unique())` will then give you the number of unique values in a vector, equivalent to `n_distinct()` in the tidyverse.
  - You can also extract frequencies of all unique values into a `table()`:

```
table(df$CNT)
##
##  ALB  ARE  ARG  AUS  AUT  BEL  BGR  BIH  BLR  BRA  BRN  CAN  CHE  CHL  COL
CRI
##  327  755  455  763  291  288  197  213  234  597   55  821  228  254  247
205
##  CZE  DEU  DNK  DOM  ESP  EST  FIN  FRA  GBR  GEO  GRC  HKG  HRV  HUN  IDN
IRL
##  333  223  348  235 1089  230  214  252  471  321  242  152  183  238  397
157
##  ISL  ISR  ITA  JOR  JPN  KAZ  KOR  KSV  LBN  LTU  LUX  LVA  MAC  MAR  MDA
MEX
##  142  174  542  313  183  616  188  211  313  362   44  308   45  179  236
286
##  MKD  MLT  MNE  MYS  NLD  NOR  NZL  PAN  PER  PHL  POL  PRT  QAT  QAZ  QCI
```

```
QMR
##  117   50   61  191  156  251  192  253  340  187  240  276  188  197  361
61
##  QRT  ROU  RUS  SAU  SGP  SRB  SVK  SVN  SWE  TAP  THA  TUR  UKR  URY  USA
VNM
##  239  170  263  234  166  187  376  345  223  192  290  186  250  189  164
151
```

Question: How would you extract this table into Excel?

Solution:

```
tmp <- as.data.frame(table(df$CNT))

rio::export(tmp, "data/CNT.xlsx")
```

5.4 Group data

Creating summary statistics by group is another routine task. This is accommodated in the tidyverse using the `group_by()`. In base R, you can use `aggregate()`.

The arguments, in addition to the data set, are simply the grouping variables separated by commas.
For example, let's calculate the mean per gender:

```
# With base R:
aggregate(wage ~ sex, wages, mean)
##      sex     wage
## 1 female 5.146924
## 2   male 6.313021
# With the tidyverse
wages %>% group_by(sex) %>% summarize(avg.wage = mean(wage))
## # A tibble: 2 x 2
##   sex    avg.wage
##   <fct>     <dbl>
## 1 female     5.15
## 2 male       6.31
```

About the `aggregate` function, note the `~` to separate `wage` and `sex`. This means by, and is very much used in the formulas of any statistical models, as we will see later in this course.

5.5 Arrange (sort) data

If you want to sort your data by the values of a particular variable, you can do so as well with the `arrange()` function (tidyverse), or with the `sort()` function (base R).

```
# With base R, order by increasing experience and decreasing wage
wages[order(wages[, "exper"], -wages[, "wage"]), ]

# Same with the tidyverse
wages %>% arrange(exper, -wage)
```

5.6 Recode variables

Along with renaming variables, recoding variables is another integral part of data wrangling. You can do this with or without the tidyverse. For example, to recode `sex` as a dummy variable

```
# With the tidyverse
wages$is_female <- wages$sex %>% recode("male" = 0,
                                        "female" = 1)
```

```
# With base R
wages$is_female <- as.integer(wages$sex == "female")
```

5.7 Identify missing values

Does one of your variables contains missing values? You can assess if an observations is missing with the function `is.na()`. Missing values in R are coded `NA`. For example:

```
a <- NA

is.na(a)
## [1] TRUE
```

You can apply it to a vector:

```
missing_sc016Q01ta <- is.na(df$SC016Q01TA)
```

To know the number of cases are found in a categorical or a logical variable, use `table`. The table commands counts all the occurence of each values of a variables. Example:

```
table(missing_sc016Q01ta)
## missing_sc016Q01ta
## FALSE  TRUE
## 18853  3050
```

5.7.1 *Removing missing values*

In the previous lesson, we removed the missing values using:

```
df <- df[complete.cases(df), ]
```

The command above remove all the rows containing missing values on any variables of `df`.

Question: Using `is.na()`, how could you remove rows containing missing values on `SC016Q01TA`?
Solution:

```
df_2 <- df[!is.na(df$SC016Q01TA), ]
```

5.7.2 *Recode missing Values*

Another problem characteristic of observational data is missing data. In R, the way to represent missing data is with the value `NA`. You can recode missing value that *should be* `NA` but are code using a different schema either by using brackets, or the tidyverse na_if() function.

```
# Replace 99-denoted missing data with NA

# With base R
wages[wages$school == 99, ] <- NA

# With the tidyverse
wages$school <- wages$school %>% na_if(99)
```

You can check for (correctly-coded) missing-values using the `is.na()` function.

```
## Missing
table(is.na(wages$school))
##
## FALSE
##  3294
```

Note: R does not naturally support multiple types of missingness like other languages, although it's possible to use the `sjmisc` package to do this.

6 TO DO BEFORE THE NEXT CLASS

1. Go through this material again and try other cases: run all the codes for yourself, change the values, change the parameters, see what happens.
2. If you haven't finished during the class, finish the practice assignment of Module 2.

Next time, we will move on, and consider the knowledge provided here is mastered.