Manipulation de données avec dplyr

Rennes, 2016

Ewen Gallic
http://egallic.fr

Structures: Data Frames

- In Economics, this might be the most frequent structure we use
- `data.frame` objects are lists of vectors
- Each column is a vector: the mode inside each column needs to be the same of all observation
- The `data.frame()` function is used to create a `data.frame`

```
women <- data.frame(height = c(58, 59, 60, 61, 62, 63, 64, 65,
                                66, 67, 68, 69, 70, 71, 72),
                     weight = c(115, 117, 120, 123, 126, 129, 132,
                                135, 139, 142, 146, 150, 154, 159, 164))
```

Structures: Data Frames

```
head(women)
```

```
##   height weight
## 1     58    115
## 2     59    117
## 3     60    120
## 4     61    123
## 5     62    126
## 6     63    129
```

```
class(women)
```

```
## [1] "data.frame"
```

Structures: Data Frames

```
dim(women)
```

```
## [1] 15  2
```

```
nrow(women)
```

```
## [1] 15
```

```
ncol(women)
```

```
## [1] 2
```

Import Data

- Whatever the type of data, there is probably a function to import it in the R session
- With ASCII files, the two main functions are `read.table()` ans `scan()`
- We will not present the `scan()` function here
- With other type of files, one needs to load a specific library

Import Data: `read.table()`

- The `read.table()` function is designed for data already organized as a table
- The output is a `data.frame`
- Here are the main parameters I use:

ARGUMENT	DESCRIPTION
`file`	File name, or complete path to file (can be an URL)
`header`	Whether the file contains the names of the variables at its first line ? (`FALSE` by default)
`sep`	Field separator character (white character by default)
`dec`	Character used for decimal points ("`.`" by default)
`na.strings`	Character vector of strungs to be interpreded as `NA` (`NA` by default)

Import Data from Excel Files

- I mainly use two functions:
  - `read.xls()` from the `gdata` package
  - `read_excel()` from the `readxl` package
- For convenience, we will use the `iris.xls` file contained in the folder of the `gdata` package

```
library(gdata)
xlsfile <- file.path(path.package("gdata"), "xls", "iris.xls")
iris <- read.xls(xlsfile) # Creates a temporary csv file
```

- By default, the first sheet is imported. The `sheet` argument enables to import another sheet, either by giving the number or the name of the sheet
- The `read_excel()` function is faster, has almost the same names for the arguments, but is not as robust at the moment as the `read.xls()` function. In addition, it returns a `tbl_df` object, not a `data.frame`

Export Data from R

- The function `write.table()` can be used to export a `data.frame` object (or a matrix) to an ASCII file:

```
write.table(my_data_frame, file = "file_name.txt", sep = ";")
```

- To save one or more objects as is: `save()` ; to import the object(s) back: `load()`:

```
save(obj_1, obj_2, file = "my_file.rda")
load("my_file.rda")
```

- To save the entire session: `save.image()`; to load the session: `load()`

```
save.image("my_session.rda")
load("my_session.rda")
```

Access elements of a vector

- Elements of a vector can be accessed by their **numerical index** or by their **name** (if they are provided with one)
- This can be done by the `"["()` function
- The arguments of this function are the vector one wants to extract data from and a numerical vector which contains the positions of the elements one wants to extract (or not), or a logical vector (mask)
- As it might be painful to write this function, R provides a shortcut to use the `"["()` function:

```
x <- c(4, 7, 3, 5, 0)
"["(x, 2)
```

```
## [1] 7
```

Access elements of a vector

```
x[2] # The second element of x
```

```
## [1] 7
```

```
x[-2] # All the elements of x minus the second one
```

```
## [1] 4 3 5 0
```

```
x[3:5] # Elements of x from 3rd to 5th position
```

```
## [1] 3 5 0
```

Access elements of a vector

```
i <- 3:5 ; x[i] # Elements of x from 3rd to 5th position
```

```
## [1] 3 5 0
```

```
x[c(F, T, F, F, F)] # Second element from x
```

```
## [1] 7
```

```
x[x<1] # Elements of x that are lower than 1
```

```
## [1] 0
```

```
x<1 # Returns a logical vector
```

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

Access elements of a vector

- To extract the positions of `TRUE` values from a logical vector: `which()`
- To extract the positions of the first minimum (maximum) of a logical or numerical vector: `which.min()` (`which.max()`)

```
x <- c(2, 4, 5, 1, 7, 6)
which(x < 7 & x > 2)
```

```
## [1] 2 3 6
```

```
which.min(x)
```

```
## [1] 4
```

Access elements of a vector

```
which.max(x)
```

```
## [1] 5
```

```
x[which.max(x)]
```

```
## [1] 7
```

Modify elements of a vector

- Simply use the `<-` symbol

```
x <- seq_len(5)
x[2] <- 3
x
```

```
## [1] 1 3 3 4 5
```

- Multiple elements can be modified using one instruction

```
x[2] <- x[3] <- 0
x
```

```
## [1] 1 0 0 4 5
```

Access elements of a matrix or data.frame

- The same function `"["()` works
- One just needs to indicate the rows (`i`) and columns (`j`) indices: `x[i,j]`

```
(x <- matrix(1:9, ncol = 3, nrow = 3))
```

```
##      [,1] [,2] [,3]
## [1,]    1    4    7
## [2,]    2    5    8
## [3,]    3    6    9
```

```
x[1, 2]
```

```
## [1] 4
```

Access elements of a matrix or data.frame

- `i` and `j` can be vectors of length greater than one:

```
i <- c(1,3) ; j <- 3
x[i,j]  # Elements of first and third row for the third column
```

```
## [1] 7 9
```

- Not providing `i` returns all lines for the `j` columns
- Not providing `j` returns all columns for the `i` rows

```
x[, 2] # Elements of the second column
```

```
## [1] 4 5 6
```

Access elements of a matrix or data.frame

- As for vectors, negative values indicate positions one does not want:

```
x[, -c(1,3)]  # x without first and third columns
```

```
## [1] 4 5 6
```

Access elements of a matrix or data.frame

- In the case of a `data.frame`, columns are named and can thus be accessed using these names

```
women <-data.frame(height =c(58, 59, 60, 61, 62, 63, 64,
                              65, 66, 67, 68,69, 70, 71, 72),
                    weight =c(115, 117, 120, 123, 126, 129, 132, 135,
                              139,142, 146, 150, 154, 159, 164))
colnames(women) # Names of the columns
```

```
## [1] "height" "weight"
```

```
rownames(women) # Names of the rows
```

```
##  [1] "1"  "2"  "3"  "4"  "5"  "6"  "7"  "8"  "9"  "10" "11" "12" "13" "14"
## [15] "15"
```

Access elements of a matrix or data.frame

```r
dimnames(women) # Names of both rows and columns
```

```
## [[1]]
##  [1] "1"  "2"  "3"  "4"  "5"  "6"  "7"  "8"  "9"  "10" "11" "12" "13" "14"
## [15] "15"
## 
## [[2]]
## [1] "height" "weight"
```

Access elements of a matrix or data.frame

- To access a specific column: `$` :

```
women$height
```

```
##  [1] 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72
```

Data manipulation with dplyr

- The packeg `dplyr` offers many functions that are really easy to use to manipulate data
- We will also use the pipe (`%>%`) operator (from the package `magrittr`), which transmits a value as the first argument of the following function
- For instance :

```
library(magrittr)
mean(x) %>% log()
```

- Computes the mean of the object `x` and the apply the logarithm function to the result of `mean(x)`. It can also be written in the following (but harder to read) way:

```
log(mean(x))
```

```
## [1] 1.609438
```

Data manipulation with dplyr: selection

- To select columns from a `data.frame`: `select()`

```
library(dplyr)
women %>%
  select(height)
```

Data manipulation with dplyr: selection

- To remove a columns from a `data.frame`: `select()` and a negative sign

```
library(dplyr)
women %>%
  select(-height) %>%
  head()
```

```
##   weight
## 1    115
## 2    117
## 3    120
## 4    123
## 5    126
## 6    129
```

Data manipulation with dplyr: selection

- To select rows according to their position: `slice()`

```
women %>% slice(4:5)
```

```
##   height weight
## 1     61    123
## 2     62    126
```

Data manipulation with dplyr: filtering

- To return rows with matchin conditions: `filter()`

```
women %>%
  filter(height == 60)
```

```
##   height weight
## 1     60    120
```

```
women %>%
  filter(weight > 120, height <= 62)
```

```
##   height weight
## 1     61    123
## 2     62    126
```

Data manipulation with dplyr: column modifications

- To rename a column: `rename(data, new_name_1 = old_name_1, new_name_2 = old_name_2)`

```
women <-
  women %>%
  rename(masse = weight)
head(women)
```

```
##   height masse
## 1     58   115
## 2     59   117
## 3     60   120
## 4     61   123
## 5     62   126
## 6     63   129
```

Data manipulation with dplyr: column modifications

- Let us create another `data.frame`:

```
unemp <- data.frame(year = 2012:2008,
                    unemployed = c(2.811, 2.604, 2.635, 2.573, 2.064),
                    active_pop = c(28.328, 28.147, 28.157, 28.074, 27.813))
```

Data manipulation with dplyr: column modifications

- To modify (or create) columns: `mutate()`

```
unemp <-
  unemp %>%
  mutate(unemp_rate = unemployed/active_pop*100,
         log_unemployed = log(unemployed),
         year = year / 1000)
head(unemp)
```

```
##    year unemployed active_pop unemp_rate log_unemployed
## 1 2.012      2.811     28.328   9.923044      1.0335403
## 2 2.011      2.604     28.147   9.251430      0.9570487
## 3 2.010      2.635     28.157   9.358241      0.9688832
## 4 2.009      2.573     28.074   9.165064      0.9450725
## 5 2.008      2.064     27.813   7.420990      0.7246458
```

Data manipulation with dplyr: ordering

- Let us create another `data.frame`:

```
df <- data.frame(last_name = c("Durand", "Martin",
                               "Martin", "Martin", "Durand"),
                 first_name = c("Sonia", "Serge", "Julien-Yacine",
                                "Victor", "Emma"),
                 grade = c(23, 18, 17, 17, 19))
```

Data manipulation with dplyr: ordering

- To order observations according to one or multiple values: `order()`:

```
df %>% arrange(first_name, last_name)
```

```
##   last_name    first_name grade
## 1    Durand          Emma    19
## 2    Martin Julien-Yacine    17
## 3    Martin         Serge    18
## 4    Durand         Sonia    23
## 5    Martin        Victor    17
```

- To order by decreasing values: `desc()` (negative sign can be used for numeric columns)

```
df %>% arrange(first_name, desc(last_name))
```

Data manipulation with dplyr: joining two `data.frame`

- Functions to join `data.frames` from `dplyr` have an easy syntax:

```
xxx_join(x, y, by = NULL, copy = FALSE, ...)
```

- `x` and `y` are the two tables to join
- `by` is a character vector containing variables used to join the tables (if ommited, a natural join using all variables with common names accross the two tables will be done)

Data manipulation with dplyr: joining two `data.frame`

- Let us create two `data.frame` to illustrate the different join functions:

```
exportations <- data.frame(year = 2011:2013,
                           exportations = c(572.6, 587.3, 597.8))
importations <- data.frame(annee = 2010:2012,
                           importations = c(558.1, 625.3,628.5))
```

Data manipulation with dplyr: joining two `data.frame`

- `inner_join()`: return all rows from `x` where there are matching values in `x`, and all columns from `x` and `y`. If there are multiple matches between `x` and `y`, all combination of the matches are returned

```
exportations %>%
  inner_join(importations, by = c(year = "annee"))
```

```
##   year exportations importations
## 1 2011        572.6        625.3
## 2 2012        587.3        628.5
```

Data manipulation with dplyr: joining two `data.frame`

- `left_join()`: return all rows from `x`, and all columns from `x` and `y`. Rows in `x` with no match in `y` will have `NA` values in the new columns. If there are multiple matches between `x` and `y`, all combinations of the matches are returned

```
exportations %>%
  left_join(importations, by = c(year = "annee"))
```

```
##   year exportations importations
## 1 2011        572.6        625.3
## 2 2012        587.3        628.5
## 3 2013        597.8           NA
```

Data manipulation with dplyr: joining two `data.frame`

- `right_join()`: return all rows from `y`, and all columns from `x` and `y`. Rows in `y` with no match in `x` will have `NA` values in the new columns. If there are multiple matches between `x` and `y`, all combinations of the matches are returned

```
exportations %>%
  right_join(importations, by = c(year = "annee"))
```

```
##   year exportations importations
## 1 2010           NA        558.1
## 2 2011        572.6        625.3
## 3 2012        587.3        628.5
```

Data manipulation with dplyr: joining two `data.frame`

- `semi_join()`: return all rows from `x` where there are matching values in `y`, keeping just columns from `x`

```
exportations %>%
  semi_join(importations, by = c(year = "annee"))
```

```
##   year exportations
## 1 2011        572.6
## 2 2012        587.3
```

Data manipulation with dplyr: joining two `data.frame`

- `anti_join()`: return all rows from `x` where there are not matching values in `y`, keeping just columns from `x`.

```
exportations %>%
  anti_join(importations, by = c(year = "annee"))
```

```
##   year exportations
## 1 2013        597.8
```

Data manipulation with dplyr: joining two `data.frame`

- `full_join()`: return all rows and all columns from both `x` and `y`. Where there are not matching values, returns `NA` for the one missing

```
exportations %>%
  full_join(importations, by = c(year = "annee"))
```

```
##   year exportations importations
## 1 2011        572.6        625.3
## 2 2012        587.3        628.5
## 3 2013        597.8           NA
## 4 2010           NA        558.1
```

Data manipulation with dplyr: aggregation

- To aggregate data, `dplyr` offers an easy way: `summarise()`
- The arguments are a `data.frame` and one or multiple operations to do on the `data.frame`
- Let us create some dummy observations:

```
# Nombre d'ingenieurs et cadres au chômage
chomage <- data.frame(region = rep(c(rep("Bretagne", 4),
                                     rep("Corse", 2)), 2),
                      departement = rep(c("Cotes-d'Armor", "Finistere",
                                          "Ille-et-Vilaine", "Morbihan",
                                          "Corse-du-Sud", "Haute-Corse"), 2),
                      annee = rep(c(2011, 2010), each = 6),
                      ouvriers = c(8738, 12701, 11390, 10228, 975, 1297,
                                   8113, 12258, 10897, 9617, 936, 1220),
                      ingenieurs = c(1420, 2530, 3986, 2025, 259, 254,
                                     1334, 2401, 3776, 1979, 253, 241))
```

Data manipulation with dplyr: aggregation

- If we want to compute the mean and standard deviation for the colums `ouvriers` and `ingenieurs`:

```
chomage %>%
  summarise(moy_ouvriers = mean(ouvriers),
            sd_ouvriers = sd(ouvriers),
            moy_ingenieurs = mean(ingenieurs),
            sd_ingenieurs = sd(ingenieurs))
```

```
##   moy_ouvriers sd_ouvriers moy_ingenieurs sd_ingenieurs
## 1     7364.167    4801.029       1704.833      1331.482
```

Data manipulation with dplyr: aggregation

- It is really simple to aggregate data on groups of observations, thanks to the `group_by()` function
- We just need to first group the data according to some values taken by one or multiple variables, and then apply the aggregation to the result:

```
chomage %>%
  group_by(annee) %>%
  summarise(ouvriers = sum(ouvriers),
            ingenieurs = sum(ingenieurs))
```

```
## # A tibble: 2 × 3
##   annee ouvriers ingenieurs
##   <dbl>    <dbl>      <dbl>
## 1  2010    43041       9984
## 2  2011    45329      10474
```

Data manipulation with dplyr: aggregation

- With groups depending on combination of variables:

```
chomage %>%
  group_by(annee, region) %>%
  summarise(ouvriers = sum(ouvriers),
            ingenieurs = sum(ingenieurs))
```

```
## Source: local data frame [4 x 4]
## Groups: annee [?]
##
##   annee   region ouvriers ingenieurs
##   <dbl>   <fctr>    <dbl>      <dbl>
## 1  2010 Bretagne    40885       9490
## 2  2010    Corse     2156        494
## 3  2011 Bretagne    43057       9961
## 4  2011    Corse     2272        513
```

Data manipulation: tidyr

- The package `tidyr` contains interesting functions to manipulate data
- These functions are really important when one realise graphs with ggplot2
- Unfortunately, their use is not as straightforward as the functions from the `dplyr` package
- We will only focus on two functions here: `gather()` and `spread()`
- These functions are useful to turn a large table to a long one, and reciprocally

Data manipulation: from a large table to a long one

- First, let us create some dummy data:

```
pop <- data.frame(city = c("Paris", "Paris", "Lyon", "Lyon"),
                  arrondissement = c(1, 2, 1, 2),
                  pop_municipale = c(17443, 22927, 28932, 30575),
                  pop_all = c(17620, 23102, 29874, 31131))
```

Data manipulation: from a large table to a long one

- The `gather()` function takes a `data.frame` as its first argument
- The second argument (`key`) is the name we want to give to the column that will contain the the names of the columns we want to gather, as a factor
- The third argument (`value`) is the name we want to give to the column that will contain the corresponding values
- Then, we need to specify which colums to gather (either by giving or excluding variable names, as in the `select()` function)

Data manipulation: from a large table to a long one

```
library(tidyr)
pop_long <-
  pop %>%
  gather(key = type_pop,
         value = population,
         pop_municipale,pop_all)
pop_long
```

```
##    city arrondissement       type_pop population
## 1 Paris              1 pop_municipale      17443
## 2 Paris              2 pop_municipale      22927
## 3  Lyon              1 pop_municipale      28932
## 4  Lyon              2 pop_municipale      30575
## 5 Paris              1        pop_all      17620
## 6 Paris              2        pop_all      23102
## 7  Lyon              1        pop_all      29874
## 8  Lyon              2        pop_all      31131
```

Data manipulation: from a long table to large one

- Now to go from a long table to a large one: `spread()`
- The first argument is the `data.frame`
- The second argument is the name of the colum that contains values that can be converted to a factor. Each level of the factor will end up as a column name
- The third argument is the name of the column that contains the values

Data manipulation: from a long table to large one

```
pop_long %>%
  spread(type_pop, population)
```

```
##    city arrondissement pop_all pop_municipale
## 1  Lyon              1   29874          28932
## 2  Lyon              2   31131          30575
## 3 Paris              1   17620          17443
## 4 Paris              2   23102          22927
```