Introduction to R, week 6

Indrek Seppo

Dec 8, 2020

HITSA Preparation of this course was supported by HITSA – Estonian Information Technology Foundation for Education.

Lets download the original piaac again:

```
piaac <- read.csv("http://www.ut.ee/~iseppo/piaacENG.csv")
```

Lists

We have talked about different data structures in R – mainly `vectors` and `data frames`, but there are actually more. One of the most important ones is a `list`. Lists are a the most versatile data structures in R. They can contain anything. Think of them as an whole excel file – you can have a number of sheets in this file, every sheet can contain a number of tables or single values or graphs.

Lists are important in R as most of the results from different analysis performed by R are returned as lists. And while you can usually take a peak at these results by either saying `summary(result)` or just printing the `result`, there is much more inside. Take a look at `t.test()`:

```
t.test(piaac$Numeracy)
```

If we write this result into a data object and look into it, we will see that it is in fact a list!

```
result <- t.test(piaac$Numeracy)
```

```
str(result)
```

There is something called `conf.int` inside this list! Lets take a look. To access the stuff inside a list you will need to use double square brackets: `listname[["objectname"]]`[1]:

[1] You will get something out using the single brackets as well, but this will be a list again, not a dataframe, vector or single number that you are usually after.

```
result$conf.int
```

```
## [1] 276.1748 278.3219
## attr(,"conf.level")
## [1] 0.95
```

This is a vector (don't mind the attributes)! To access the first value:

```r
result$conf.int[1]
```

```
## [1] 276.1748
```

And to store this value in an data object you can use later:

```r
lower <- result$conf.int[1]
```

You can create your own lists:

```r
name <- "Masha"
fingers <- c(5, 5)
Masha <- list(name=name, nr_of_fingers=fingers)

Masha
```

```
## $name
## [1] "Masha"
##
## $nr_of_fingers
## [1] 5 5
```

```r
Masha$name
```

```
## [1] "Masha"
```

```r
Masha[["name"]]
```

```
## [1] "Masha"
```

Note that there is one peculiar thing with lists. You can add variables with the same name into list, and it only outputs you the first value, if you access it by name:

```r
somelist <- list(a=3, b=4, a=5)
somelist$a
```

```
## [1] 3
```

Wide and long data

Lets recap the data in wide and long format again. In a previous session we took a quick look at `pivot_wider()` and `pivot_longer()` from a package called `tidyr()` to convert from wider to longer format and back.

Lets try it with the same `animals` dataset we have just downloaded. If not, download it:

```
animals <- read.csv("http://www.ut.ee/~iseppo/animals.csv")
```

Take a look at the help file of `pivot_longer()`. This is a rather powerful function with a nr of arguments, but the main ones are: `data` – which dataset to convert, `cols` – which columns to pivot into longer format, `names_to` – what should be the name of the new variable containing these values, `values_to` – what should be the name of the new variable containing the values which are currently inside the matrix.

Lets try it:

```
animals_long <- pivot_longer(animals, cols=c("length", "width", "age"),
                             names_to="measurement", values_to="value")
```

If you have data in long format, then you can widen it with `pivot_wider()`. Again, lets look at the help file: `?pivot_wider()`. It requires the data frame on which to operate, then the `id_cols` – these need to identify each observation uniquely (you will notice if there are problems with this), by default you do not have to put anything there, all of the other columns you are not touching, will just be left in place. What is important for our simple use cases are `names_from` and `values_from` parameters stating which columns should be moved to variable names and how to populate the data matrix below it.

Lets take a look:

```
pivot_wider(data=animals_long, names_from="measurement", values_from="value")
```

```
## # A tibble: 2 x 4
##   animal length width   age
##   <chr>   <int> <int> <int>
## 1 bear       10     7     3
## 2 cat         4     3     5
```

And it is done!

There is also a nice vignette about these functions available here: https://tidyr.tidyverse.org/articles/pivot.html

Lets try to use these functions with our data. First lets create some aggregated data together:

```
aver_num_and_wage <- piaac %>%
  group_by(studyarea)%>%
  summarize(average_num=mean(Numeracy, na.rm=T),
            average_wage=mean(Income, na.rm=T))
```

```
## `summarise()` ungrouping output (override with `.groups` argument)
```

Take a look at the data.

Your turn:

- Convert this dataset to a longer format, which would look like this:

```
## # A tibble: 4 x 3
##   studyarea                                    measurement  values
##   <chr>                                        <chr>         <dbl>
## 1 Agriculture and veterinary                   average_num    272.
## 2 Agriculture and veterinary                   average_wage   746.
## 3 Engineering, manufacturing and construction average_num    275.
## 4 Engineering, manufacturing and construction average_wage   995.
```

- Starting now from this long format that you just created, convert it so that the studyareas would be the columns, and average income and average numeracy will be the rows.

Joining data

In the real life you do not have data in one dataset. You have multiple datasets you would like to join with each other. `dplyr` offers functions for just that. We have `left_join()`, `right_join()`, `inner_join()`, `full_join()` `anti_join()` and `semi_join()`. Lets take a look at them. Lets create couple of small datasets:

```
animallength <- data.frame(animal=c("cat", "dog", "elephant"), length=c(10, 20, 50))
animalwidth <- data.frame(animal=c("cat", "dog", "bear"), width=c(5, 15, 10))
```

And lets try it:

```
animals <- left_join(animallength, animalwidth)
animals
```

```
##     animal length width
## 1      cat     10     5
## 2      dog     20    15
## 3 elephant     50    NA
```

It took all the animals from the first dataset (the dataset on the left) and added the values from the second dataset to these. If it could not find a matching animal from the second dataset, it added a `NA`.

By default it tries to merge the datasets by the variables present in both datasets. In case the variable names are different in different datasets (*e.g.* Company in the first one and Firm in the second one) you can specify this with a parameter `by=c("Company", "Firm")`.

The `right_join()` would take all the animals in the second dataset and try to add columns from the first dataset to these[2].

'full_join() would retain all the animals from both dataset:

[2] Why would you need both `left_join()` and `right_join()` if you could just switch the arguments? The reason is the piping in tidyverse – if you use `%>%` the result of the previous lines of code will be presented as the first argument to `right_join()` or `left_join()`.

```r
full_join(animallength, animalwidth)
```

```
##     animal length width
## 1      cat     10     5
## 2      dog     20    15
## 3 elephant     50    NA
## 4     bear     NA    10
```

And `inner_join()` would only leave in the animals which are present on both datasets:

```r
inner_join(animallength, animalwidth)
```

```
##   animal length width
## 1    cat     10     5
## 2    dog     20    15
```

`anti_join()` would compare the datasets and remove everything that is present in both dataset, leaving only the rows which are in the first dataset and not in the second one (you'll actually find use to this – this is a quick way for testing whether there are some values in one dataset that are missing in another).

```r
anti_join(animallength, animalwidth)
```

```
##     animal length
## 1 elephant     50
```

```r
semi_join(animallength, animalwidth)
```

```
##   animal length
## 1    cat     10
## 2    dog     20
```

Your turn:

First read in the the following datasets: `http://www.ut.ee/~iseppo/gdppercap.csv` and `http://www.ut.ee/~iseppo/population.csv`. But use `read_csv()` from `readr` package. We have used base-R-s `read.csv()` mostly in the class[3], but readr is in fact a better option. It does some things differently from `read_csv()`: it does not convert text to factors automatically (you have to do it yourself manually by `df$variable <- as.factor(df$variable)`), it retains the original column names if they contain spaces or other special

[3] The reason being that at one point `readr` had a bug and did not work well with internet addresses – this is fixed now.

characters (`read.csv()` replaces them with points), it even tries to parse the dates and convert them to date-type variables. It also reads zipped files and it is much quicker than `read.csv()`. Note that expects the files to be in UTF-8 encoding by default.

- take a look at these files – which variables do they contain? Now join them so that all the countries present in either of the datset would be included. For how many countries is there no data of GDP per capita for this year?

You might also want to add some datasets "on top of each other". Base-R offers a function called `rbind()` for this (and `cbind` to bind together columns), dplyr has `bind_rows()` – a somewhat more intelligent version. Download the following files:

Lets take a look how Estonian companies are doing. Go the the site https://www.emta.ee/eng/taxes-paid and download the data of every Estonian company for the first three quarters of 2018 in Excel format.

Lets read them in now, using `read_excel()` function from `readxl` package:

```
library(readxl)
q1.2018 <- read_excel("tasutud_maksud_2018_i_kvartal_eng.xlsx")
q2.2018 <- read_excel("tasutud_maksud_2018_ii_kvartal_eng.xlsx")
q3.2018 <- read_excel("tasutud_maksud_2018_iii_kvartal_eng.xlsx")
```

Take a look at them. These are all the business entities in Estonia detailing how many people they employed, how much did thay pay VAT and how much employment taxes. Estonian tax and customs board gives this kind of data out every quarter and you can download it from their website.

What could we do with this data? Lets add it together to one big dataframe! But to be able to tell the difference – from which quarter was the data, we will need to add the information to the datasets:

```
q1.2018$quarter <- "q1.2018"
q2.2018$quarter <- "q2.2018"
q3.2018$quarter <- "q3.2018"
```

Now we can create a single dataset:

```
tax.2018 <- bind_rows(q1.2018, q2.2018, q3.2018)
dim(tax.2018)
```

```
## [1] 383361     11
```

If we take a closer look, then we notice that there is actually a problem with the data. For some quarters the counties are with capital C and with some with small c (Harju County vs Harju county). I am going to change them all to have small c now:

```
library(stringr)
tax.2018$County <- str_replace(tax.2018$County, "county", "County")
```

Your turn:
We have been using `group_by()` and `summarize()` a lot.

- Can you find the nr of employees (`Number of employees`) by county (`County` in the dataset - NB!, this is county, not country) and quarter using these two verbs? Use `na.rm=T` in the `sum()` as a parameter

```
## `summarise()` regrouping output by 'County' (override with `.groups` argument)
```

```
## # A tibble: 3 x 3
## # Groups:   County [1]
##   County       quarter nrofemployees
##   <chr>        <chr>           <dbl>
## 1 Harju County q1.2018        401086
## 2 Harju County q2.2018        409192
## 3 Harju County q3.2018        409822
```

We were using `pivot_wider()` from the package `tidyr` to convert tidy data to wide format. Load the package `tidyr`, look at the help text of `pivot_wider()` and widen the data so that the name of the quarter becomes a new variable and the `nrofemployees` will be the thing that will be presented in the cells. Remember – you had to use `names_from` (from which variable will come the new variable names) and `values_from` (from which variable will come the values) parameters. The result should look something like this:

```
## # A tibble: 3 x 4
## # Groups:   County [3]
##   County           q1.2018 q2.2018 q3.2018
##   <chr>              <dbl>   <dbl>   <dbl>
## 1 Harju County      401086  409192  409822
## 2 Hiiu County         3001    3171    3043
## 3 Ida-Viru County    42625   43267   43206
```

- Create a new variable in this dataset that you just created called change, and make it so that it would be the difference between `q3.2018` and `q1.2018`. Arrange the dataset by this variable. Which counties created the most jobs and which ones the least?

Factors

We learned that for dealing with factors there is a package called `forcats`. To manually change the order of factor levels, you can use `fct_relevel()`.

For example: the current order of the `piaac$Health` (if it would be cast to a factor) would be:

```
levels(as.factor(piaac$Health))
```

```
## [1] "Excellent" "Fair"      "Good"      "Poor"      "Very good"
```

Lets change it:

```
library(forcats)
piaac <- piaac %>%
  mutate(Health = fct_relevel(Health, "Poor", "Fair", "Good", "Very good", "Excellent"))
levels(piaac$Health)
```

```
## [1] "Poor"      "Fair"      "Good"      "Very good" "Excellent"
```

But you could also have done it with a one-liner:

```
piaac$Health = fct_relevel(piaac$Health, "Poor", "Fair", "Good", "Very good", "Excellent")
```

To change the factor levels, you can use `fct_recode()`:

```
unique(piaac$children)
```

```
## [1] "Has children" "No children"  NA
```

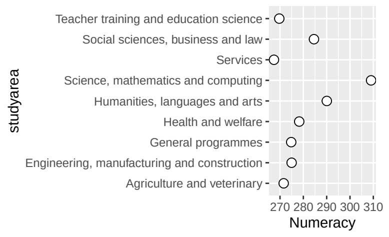
```
piaac <- piaac %>%
  mutate(children=fct_recode(children, "Yes"= "Has children", "No"="No children"))
levels(piaac$children)
```

```
## [1] "Yes" "No"
```

Third thing you would do quite a lot is order the level of factors by some variable. Lets create again a simple summarizing table of average numeracy scores by study area:

```
numscores <- piaac %>%
  group_by(studyarea)%>%
  summarize(Numeracy = mean(Numeracy))

ggplot(numscores, aes(x=Numeracy, y=studyarea))+
  geom_point(shape=21, size=3, fill="white")
```



These are not in a nice order at all. What can we do to plot them so that the highest would be on top and the lowest in the bottom? We can change the order of level manually, but there is an easier way. We can use `fct_reorder()` from `forcats`:

```
numscores$studyarea <- fct_reorder(numscores$studyarea, numscores$Numeracy)
```

Now try to run the plot again!

Your turn:

- Take the piaac dataset and change the levels of factor in Education variable so that “Higher” will become “High”, “Basic” to “Low” and “Secondary” to “Medium”. Use the **fct_recode()** function from the **forcats** package. The example from the help section will show you how to do it.
- Change the order of Education factor levels so that it will be Low, Medium, High. Use the **fct_relevel()** function from the forcats package to do it. The example in the help section will show you how to do it.