Practical 204

Stefano De Sabbata

2020-11-30

Data wrangling Pt. 1

Stefano De Sabbata

R Projects

RStudio provides an extremely useful functionality to organise all your code and data, that is **R Projects**. Those are specialised files that RStudio can use to store all the information it has on a specific project that you are working on – *Environment*, *History*, working directory, and much more, as we will see in the coming weeks.

In RStudio Server, in the *Files* tab of the bottom-left panel, click on *Home* to make sure you are in your home folder – if you are working on your own computer, create a folder for these practicals wherever most convenient. Click on *New Folder* and enter *Practicals* in the prompt dialogue, to create a folder named *Practicals*.

Select *File > New Project …* from the main menu, then from the prompt menu, *New Directory*, and then *New Project*. Insert *Practical_204* as the directory name, and select the *Practicals* folder for the field *Create project as subdirectory of*. Finally, click *Create Project*.

RStudio has now created the project, and it should have activated it. If that is the case, the *Files* tab in the bottom-right panel should be in the *Practical_204* folder, which contains only the *Practical_204.Rproj* file. The *Practical_204.Rproj* stores all the *Environment* information for the current project and all the project files (e.g., R scripts, data, output files) should be stored within the *Practical_204* folder. Moreover, the *Practical_204* is now your working directory, which means that you can refer to a file in the folder by using only its name and if you save a file that is the default directory where to save it.

On the top-right corner of RStudio, you should see a blue icon representing an R in a cube, next to the name of the project (*Practical_204*). That also indicates that you are within the *Practical_204* project. Click on *Practical_204* and select *Close Project* to close the project. Next to the R in a cube icon, you should now see *Project: (None)*. Click on *Project: (None)* and select *Practical_204* from the list to reactivate the *Practical_204* project.

With the *Practical_204* project activated, select from the top menu *File > New File > R Script*. That opens the embedded RStudio editor and a new empty R script folder. Copy the two lines below into the file. The first loads the `tidyverse` library, whereas the second loads another library that the code below uses to produce well-formatted tables.

```
library(tidyverse)
library(knitr)
```

From the top menu, select *File > Save*, type in *My_script_Practical_204.R* (make sure to include the underscore and the *.R* extension) as *File name*, and click *Save*.

Install libraries

RStudio and RStudio Server come with a number of libraries already pre-installed. However, you might find yourself in the position of wanting to install additional libraries to work with.

The remainder of this practical requires the library `nycflights13`. To install it, select *Tools > Install Packages...* from the top menu. Insert `nycflights13` in the *Packages (separate multiple with space or comma)* field and click install. RStudio will automatically execute the command `install.packages("nycflights13")` (so, no need to execute that yourself) and install the required library.

As usual, use the function `library` to load the newly installed library.

```
library(nycflights13)
```

The library `nycflights13` contains a dataset storing data about all the flights departed from New York City in 2013. The code below, loads the data frame `flights` from the library `nycflights13` into the variable `flights_from_nyc`, using the `::` operator to indicate that the data frame `flights` is situated within the library `nycflights13`.

```
flights_from_nyc <- nycflights13::flights
```

Add both lines above to your R script, as well as the code snippets provided as an example below.

Loading R scripts

It is furthermore possible to load the function(s) defined in one script from another script – in a fashion similar to when a library is loaded. Create a new R script named `Practical_204_RS_functions.R`, copy the code below in that R script and save the file

```
cube_root <- function (input_value) {
   result <- input_value ^ (1 / 3)
   result
}
```

Create a second R script named `Practical_204_RS_main.R`, copy the code below in that second R script and save the file.

```
source("Practical_204_RS_functions.R")

cube_root(27)
```

Executing the `Practical_204_RS_main.R` instructs the interpreter first to run the `Practical_204_RS_functions.R` script, thus creating the `cube_root` function, and then invoke the function using `27` as an argument, thus returning again `3`. That is a simple example, but this can be an extremely powerful tool to create your own library of functions to be used by different scripts.

Data manipulation

The analysis below uses the [dplyr](https://dplyr.tidyverse.org/) library (also part of the Tidyverse), which it offers a grammar for data manipulation.

For instance, the function `count` can be used to count the number rows of a data frame. The code below provides `flights_from_nyc` as input to the function `count` through the pipe operator, thus creating a new `tibble` with only one row and one column.

As discussed in the previous lecture, a `tibble` is data type similar to data frames, used by all the Tidyverse libraries.

All Tidyverse functions output `tibble` rather than `data.frame` objects when representing a table. However, `data.frame` object can be provided as input, as they are automatically converted by Tidyverse functions before proceeding with the processing steps.

In the tibble outputted by the count function below, the column n provides the count. The function kable of the library knitr is used to produce a well-formatted table.

```
flights_from_nyc %>%
  dplyr::count() %>%
  knitr::kable()
```

n
336776

The example above already shows how the **pipe operator** can be used effectively in a multi-step operation.

The function count can also be used to count the number rows of a table that have the same value for a given column, usually representing a category.

In the example below, the column name origin is provided as an argument to the function count, so rows representing flights from the same origin are counted together – EWR is the Newark Liberty International Airport, JFK is the John F. Kennedy International Airport, and LGA is LaGuardia Airport.

```
flights_from_nyc %>%
  dplyr::count(origin) %>%
  knitr::kable()
```

origin	n
EWR	120835
JFK	111279
LGA	104662

As you can see, the code above is formatted in a way similar to a code block, although it is not a code block. The code goes to a new line after every %>%, and space is added at the beginning of new lines. That is very common in R programming (especially when functions have many parameters) as it makes the code more readable.

Summarise

To carry out more complex aggregations, the function summarise can be used in combination with the function group_by to summarise the values of the rows of a data frame. Rows having the same value for a selected column (in the example below, the same origin) are grouped together, then values are aggregated based on the defined function (using one or more columns in the calculation).

In the example below, the function sum is applied to the column distance to calculate distance_traveled_from (the total distance travelled by flights starting from each airport).

```
flights_from_nyc %>%
  dplyr::group_by(origin) %>%
  dplyr::summarise(
    distance_traveled_from = sum(distance)
  ) %>%
  knitr::kable()
```

origin	distance_traveled_from
EWR	127691515
JFK	140906931
LGA	81619161

Select and filter

The function `select` can be used to select some **columns** to output. For instance in the code below, the function `select` is used to select the columns `origin`, `dest`, and `dep_delay`, in combination with the function `slice_head`, which can be used to include only the first `n` rows (`5` in the example below) to output.

```r
flights_from_nyc %>%
  dplyr::select(origin, dest, dep_delay) %>%
  dplyr::slice_head(n = 5) %>%
  knitr::kable()
```

origin	dest	dep_delay
EWR	IAH	2
LGA	IAH	4
JFK	MIA	2
JFK	BQN	-1
LGA	ATL	-6

The function `filter` can instead be used to filter **rows** based on a specified condition. In the example below, the output of the `filter` step only includes the rows where the value of `month` is `11` (i.e., the eleventh month, November).

```r
flights_from_nyc %>%
  dplyr::select(origin, dest, year, month, day, dep_delay) %>%
  dplyr::filter(month == 11) %>%
  dplyr::slice_head(n = 5) %>%
  knitr::kable()
```

origin	dest	year	month	day	dep_delay
JFK	PSE	2013	11	1	6
JFK	SYR	2013	11	1	105
EWR	CLT	2013	11	1	-5
LGA	IAH	2013	11	1	-6
JFK	MIA	2013	11	1	-3

Notice how `filter` is used in combination with `select`. All functions in the `dplyr` library can be combined, in any other order that makes logical sense. However, if the `select` step didn't include `month`, that same column couldn't have been used in the `filter` step.

Mutate

The function `mutate` can be used to add a new column to an output table. The `mutate` step in the code below adds a new column `air_time_hours` to the table obtained through the pipe, that is the flight air time in hours, dividing the flight air time in minutes by `60`.

```r
flights_from_nyc %>%
  dplyr::select(flight, origin, dest, air_time) %>%
  dplyr::mutate(
    air_time_hours = air_time / 60
  ) %>%
  dplyr::slice_head(n = 5) %>%
  knitr::kable()
```

flight	origin	dest	air_time	air_time_hours
1545	EWR	IAH	227	3.783333
1714	LGA	IAH	227	3.783333
1141	JFK	MIA	160	2.666667
725	JFK	BQN	183	3.050000
461	LGA	ATL	116	1.933333

Arrange

The function `arrange` can be used to sort a tibble by ascending order of the values in the specified column. If the operator `-` is specified before the column name, the descending order is used. The code below would produce a table showing all the rows when ordered by descending order of air time.

```
flights_from_nyc %>%
  dplyr::select(flight, origin, dest, air_time) %>%
  dplyr::arrange(-air_time) %>%
  knitr::kable()
```

In the examples above, we have used `slice_head` to present only the first `n` (in the examples `5`) rows in a table, based on the existing order. The `dplyr` library also provides the functions `slice_max` and `slice_min` which incorporate the sorting functionality (see `slice` reference page).

As such, the following code uses `slice_max` to produce a table including only the 5 rows with the *highest* air time.

```
flights_from_nyc %>%
  dplyr::select(flight, origin, dest, air_time) %>%
  dplyr::slice_max(air_time, n = 5) %>%
  knitr::kable()
```

flight	origin	dest	air_time
15	EWR	HNL	695
51	JFK	HNL	691
51	JFK	HNL	686
51	JFK	HNL	686
51	JFK	HNL	683

The following code, instead, uses `slice_min`, thus producing a table including only the 5 rows with the *lowest* air time.

```
flights_from_nyc %>%
  dplyr::select(flight, origin, dest, air_time) %>%
  dplyr::slice_min(air_time, n = 5) %>%
  knitr::kable()
```

flight	origin	dest	air_time
4368	EWR	BDL	20
4631	EWR	BDL	20
4276	EWR	BDL	21
4619	EWR	PHL	21
4368	EWR	BDL	21
4619	EWR	PHL	21
2132	LGA	BOS	21

flight	origin	dest	air_time
3650	JFK	PHL	21
4118	EWR	BDL	21
4276	EWR	BDL	21
4276	EWR	BDL	21
4276	EWR	BDL	21
4276	EWR	BDL	21
4577	EWR	BDL	21
6062	EWR	BDL	21
3847	EWR	BDL	21

In both cases, if the table contains ties, all rows containing a value that is present among the maximum or minimum selected values are presented, as it is the case with the rows containing the value `21` in the example above.

Data manipulation example

Finally, the code below illustrates a more complex, multi-step operation using all the functions discussed above.

1. Start from the `flights_from_nyc` data.
2. Select origin, destination, departure delay, year, month, and day.
3. Filter only rows referring to flights in November.
4. Filter only rows where departure delay is not (notice that the negation operator `!` is used) `NA`.

- That is necessary because the function `mean` would return `NA` as output if any of the values in the column is `NA`.

5. Group by destination.
6. Calculated the average delay per destination.
7. Add a column with the delay calculated in hours (minutes over 60).
8. Sort the table by *descending* delay (note that `-` is used before the column name).
9. Only show the first 5 rows.
10. Create a well-formatted table.

```
flights_from_nyc %>%
  dplyr::select(origin, dest, year, month, day, dep_delay) %>%
  dplyr::filter(month == 11) %>%
  dplyr::filter(!is.na(dep_delay)) %>%
  dplyr::group_by(dest) %>%
  dplyr::summarize(
    avg_dep_delay = mean(dep_delay)
  ) %>%
  dplyr::mutate(
    avg_dep_delay_hours = avg_dep_delay / 60
  ) %>%
  dplyr::arrange(-avg_dep_delay_hours) %>%
  dplyr::slice_head(n = 5) %>%
  knitr::kable()
```

dest	avg_dep_delay	avg_dep_delay_hours
SBN	67.50000	1.1250000
BDL	26.66667	0.4444444
CAK	19.70909	0.3284848

dest	avg_dep_delay	avg_dep_delay_hours
BHM	19.61905	0.3269841
DSM	16.14815	0.2691358

Exercise 204.1

Extend the code in the script `My_script_Practical_204.R` to include the code necessary to solve the questions below.

Question 204.1.1: Write a piece of code using the pipe operator and the `dplyr` library to generate a table showing the average air time in hours, calculated grouping flights by carrier, but only for flights starting from the JFK airport.

Question 204.1.2: Write a piece of code using the pipe operator and the `dplyr` library to generate a table showing the average arrival delay compared to the overall air time (**tip**: use manipulate to create a new column that takes the result of `arr_delay / air_time`) calculated grouping flights by carrier, but only for flights starting from the JFK airport.

Question 204.1.3: Write a piece of code using the pipe operator and the `dplyr` library to generate a table showing the average arrival delay compared to the overall air time calculated grouping flights by origin and destination, sorted by destination.