R Workshop

Lecture 4: Some more advanced functionality in R and R Studio

May 14, 2024

1 Creating a Project in RStudio

The main issue that a Project is trying to solve is the issue of analysis transportability. Let's say you want to move your analysis files and data to another folder, or if you want to share it with someone else, but you open a lot of files in your R script, then you or the someone else needs to go in and change all the directories in the R script. OK, you can say, why not put a variable or `setwd()` at the beginning of the file that people can change, and yes you can do that. But, if you set up your Project correctly, you won't even need to do that, and you don't have to worry about directory information at all while writing/running the script.

To create a project, go to *File/New Project...*. If you are creating a new project, then choose "New Project", but you can also convert an existing directory into a project by choosing "Existing Project".

OK, let's continue under the assumption that you are creating a new project. It will create a folder and an `.Rproj` file in that folder. This file contains any project settings that you might opt for, but you really don't have to work with it if you don't want to. That's it. You should now store your R-script and data in the new folder.

If you want to work with the script, click on the `.Rproj` file and then go to *Files* in the plots panel in R Studio and then click on the R script that you want to work with. That's it. Everything that you do in the R script will be relativised to the folder that the `.Rproj` file is in. So, if there is a `.csv` file that you want to open, and it is in the same folder, you can just mention the file name in the `read.csv()` function. Similarly, if you save a plot and just save it with the file name, it will save to the folder with the `.Rproj` (but see the last section (Section 4) for more nuance on this).

Advanced tip: You can manually move the `.Rproj` to another folder, if you want to make it the project folder.

2 Just a clarification on `tidyverse` functions

Earlier we talked about using the `summarise()` function, but we only talked about how to summarise a single new function. But, the technique extends to multiple columns or multiple calculations quite easily. Let's use `mtcars`, which is a built-in dataset in R. Suppose you want to find the means of two separate columns (*mpg, drat*) for each gear value, you could write the code as follows:

```
mtcars %>%
  group_by(gear) %>%
  summarise(meanMPG = mean(mpg),
            meanDrat = mean(drat))
```

```
## # A tibble: 3 x 3
##    gear meanMPG meanDrat
##   <dbl>   <dbl>    <dbl>
## 1     3    16.1     3.13
## 2     4    24.5     4.04
## 3     5    21.4     3.92
```

You can also use a similar syntax to get two separate summarisations for the same column. Suppose you want to find the mean and the median for each *gear* value for the column *mpg*:

```
mtcars %>%
  group_by(gear) %>%
  summarise(meanMPG = mean(mpg),
            medianMPG = median(mpg))

## # A tibble: 3 x 3
##    gear meanMPG medianMPG
##   <dbl>   <dbl>     <dbl>
## 1     3    16.1      15.5
## 2     4    24.5      22.8
## 3     5    21.4      19.7
```

3 Advanced data munging

3.1 Merging multiple datasets

Suppose you have two sets of data with the same columns, but different values, subjects,... and want to merge the two data sets.

You can open the two data sets and then use the `bind_rows()` function from `tidyverse`. But, for it to work, both the data sets must have the same column names. If the column names are different, you will need to munge each `data.frame` separately to get them to have the same column names before combining them.

```
Data1 = read.csv("Lecture4-data1.csv")
head(Data1)

##   Sub Type   Measure1   Measure2
## 1   1    a -47.219235  -4.382947
## 2   2    b  16.153117  21.857513
## 3   3    c  -4.891709 -24.159874
## 4   4    a  91.678481  -3.006118
## 5   5    b  14.697127 -10.528346
## 6   6    c  34.863009  -2.385930

Data2 = read.csv("Lecture4-data2.csv")
head(Data2)

##   Sub Type  Measure1   Measure2
## 1 103    a  3.978076 -19.350869
## 2 104    b 10.315312   1.676916
## 3 105    c  8.210829  44.077843
## 4 106    a 17.867848  -1.728683
## 5 107    b 10.169713  31.071891
## 6 108    c 12.186301 -26.449843

#Now you can merge the two datasets into one lengthwise
FullData = Data1 %>%
  bind_rows(Data2)
```

3.2 Coding your data with more information

Suppose you have a `.csv` file, and for each measurement or each stimulus, you want to add some more information, you can just have mutate function with embedded conditionals.

```
#Modifying the data.frame directly
FullDataCoded = FullData %>%
  mutate(NewInfo = ifelse(Type == "a", "Old",
                          ifelse(Type == "b", "Recent",
                                 ifelse(Type == "c", "New",NA))))
head(FullDataCoded)

##   Sub Type   Measure1   Measure2 NewInfo
## 1   1    a -47.219235  -4.382947     Old
## 2   2    b  16.153117  21.857513  Recent
## 3   3    c  -4.891709 -24.159874     New
## 4   4    a  91.678481  -3.006118     Old
## 5   5    b  14.697127 -10.528346  Recent
## 6   6    c  34.863009  -2.385930     New
```

Advice: You can also use `case_when()`. Some people prefer using this function, but I hate learning new commands unless I have to. Try to do as much with as little as you can — it will help you practice/remember coding in R better)

But, another nice way to do it is to create a separate `data.frame` and then merge it with your data. Let's say that you want to add some more coding information about each *Type* in your data from above.

```
#New coding data.frame
coding = data.frame(Type = letters[1:3],
                    NewInfo = c("Old","Recent","New"),
                    NewInfo2 = c("Slow","Medium","Fast"))


#Now merging the new coding data.frame
FullDataCoded = FullData %>%
  full_join(coding)

## Joining with `by = join_by(Type)`

head(FullDataCoded,by=c("Type"))

##   Sub Type   Measure1   Measure2 NewInfo NewInfo2
## 1   1    a -47.219235  -4.382947     Old     Slow
## 2   2    b  16.153117  21.857513  Recent   Medium
## 3   3    c  -4.891709 -24.159874     New     Fast
## 4   4    a  91.678481  -3.006118     Old     Slow
## 5   5    b  14.697127 -10.528346  Recent   Medium
## 6   6    c  34.863009  -2.385930     New     Fast
```

In fact, if you have the coding file as a separate `.csv` file, you could just open it and then merge it with your data.

```
#new coding file
coding = read.csv("Lecture4-coding.csv")

#Now merging the new coding data.frame
```

```r
FullDataCoded = FullData %>%
  full_join(coding)

## Joining with `by = join_by(Type)`

head(FullDataCoded)

##   Sub Type   Measure1   Measure2 NewInfo NewInfo2
## 1   1    a -47.219235  -4.382947     Old     Slow
## 2   2    b  16.153117  21.857513  Recent   Medium
## 3   3    c  -4.891709 -24.159874     New     Fast
## 4   4    a  91.678481  -3.006118     Old     Slow
## 5   5    b  14.697127 -10.528346  Recent   Medium
## 6   6    c  34.863009  -2.385930     New     Fast
```

You should also look at the following functions: `inner_join()`, `left_join()`, `right_join()`.

4 Writing your own functions in R

Many times, you repeat the same chunks of code again and again in your scripts. It is useful to have the repeating code as a function that you can use. For example, maybe you want to get the same analysis and plots for different subsets of the data. In such a case, it is extremely useful to write a function that you can use repeatedly.

The basic anatomy of a function definition is similar to variable assignment. You define the arguments that the functions can take it along with the instructions within the function, and then assign the function definition a name just as if it were variable assignment.

```r
#The basic anatomy of a function
functionName = function(arg1=..., arg2=...){
  <list of instructions to excute within a function>

    return(some value of interest)
}
```

Let's say for some reason you wanted to write your own function to get the average of the values in a vector. Such a function would take a vector as an argument, and then calculate the mean and return the calculated average value.

```r
#Defining the function
AVG_Karthik = function(x){

  #Calculating the average
  average = sum(x)/length(x)

  #Returning the value as the output of the function
  return(average)
}

#Using the function
AVG_Karthik(x=1:30)

## [1] 15.5
```

```r
#Comparing our function to the pre-defined function in R
mean(x=1:30)

## [1] 15.5
```

OK, that was a silly example, but it was useful to see the anatomy of a function definition. Now, let's see if we can do a set of data processing steps and then plot something of value to us, and then save that plot. Let's say we want to work with something like the **FullData** that we had in the previous section, and we want to first filter out some condition, and then plot a facetted scatterplot for *Measure1* and *Measure1* for each *Type*, and then finally save it.

```r
#Defining the function
AnalysisAndPlotting = function(df){
  #Data processing
  dfmodified = df %>%
    filter(NewInfo2 != "Fast")

  #Creating a plot
  #I put it in parenthesis so that I can see the output when I run the function.
  plot = ggplot(dfmodified,aes(x=Measure2,y=Measure1))+
    geom_point()+facet_grid(~Type)

  ggsave("Lecture4-plot.png",plot,width=6,height=4,dpi=300)
}


#Running the function
AnalysisAndPlotting(df = FullDataCoded)
```

If you are using a Project, then the plot will get saved to the base project folder. Or you can specify a sub-folder within your Project folder for just figures, and then you would have to include the sub-folder information along with the file name.

```r
#Defining the function
AnalysisAndPlotting = function(df){
  #Data processing
  dfmodified = df %>%
    filter(NewInfo2 != "Fast")

  #Creating a plot
  #I put it in parenthesis so that I can see the output when I run the function.
  plot = ggplot(dfmodified,aes(x=Measure2,y=Measure1))+
    geom_point()+facet_grid(~Type)

  ggsave("figures/Lecture4-plot.png",plot,width=6,height=4,dpi=300)
}


#Running the function
AnalysisAndPlotting(df = FullDataCoded)
```

Advice: Make sure to have all your function definitions in one place at the top, separate from the actual data analysis. That will lead to a nicer workflow. You can also put all the function definitions in a separate r file and then use `source()` to be able to use all the functions in that file in your current file. See here for more information on this.

5 Lazy but useful

You might have noticed that some times I include the argument name with a function, and other times I don't. This is me being lazy. What I am doing is using the default semantics of R. If you input the arguments to any function, *in the order* that they are defined in the function definition, then you don't need to include the argument name. For example, the function `mean()` has multiple arguments; see Help. However, if I input the arguments in the order that they are defined in, then I don't need to specify the argument name.

```
#With argument names
mean(x=c(1:20))

## [1] 10.5

#Without argument names
mean(c(1:20))

## [1] 10.5
```

It is a really useful/clever functionality of R. But, the price to pay for the flexible semantic interpretation is that of more vigilance. If you are going to be lazy, then make sure that you have entered the arguments in the right order. Below, when the arguments are named properly, everything works well. In the second example, the second unnamed argument is of logical type but it gets assigned to *trim* which is the second argument in the function definition — this causes an error because *trim* has to be numeric. However, if the argument accidentally happens to be of the same type, then you won't get an error though you did something wrong. So be careful, if you are going to be lazy.

```
#With argument names
mean(x=c(1:20),na.rm=T)

## [1] 10.5

#Without argument names, but in the wrong order
mean(c(1:20),T)

## Error in mean.default(c(1:20), T): 'trim' must be numeric of length one
```