02 Working with data

Dr Juan H Klopper

February 21, 2019

Table of Contents

[Preamble 1](#_Toc3472280)

[Setting the working directory 2](#_Toc3472281)

[Creating computer variables to work with 3](#_Toc3472282)

[Using indexing (addressing) to select specific values 3](#_Toc3472283)

[Importing a library 4](#_Toc3472284)

[Importing a data file 5](#_Toc3472285)

[Exploring the data 5](#_Toc3472286)

[The dplyr library 7](#_Toc3472287)

[filter 7](#_Toc3472288)

[select 9](#_Toc3472289)

[mutate 11](#_Toc3472290)

[Conclusion 12](#_Toc3472291)



## Preamble

In this lecture, we will learn how to import a spreadsheet file containing data for a simulated research project (simulated in as much as the data point values for the variables were created using the sample() and the rnorm() function that we saw in the previous tutorial\_. Once imported, we can extract specific parts of our data to analyze it and thereby answer our research questions.

In order to extract this data, we need to learn about *addressing* or *indexing*. This concept simply refers to an address given to each data point value in a dataset. The same happens in a Microsoft Excel file, where every cell is named for is column and row value, i.e. *B7*, referring to the data in the cell in column *B* and in row *7*.

Along this exciting journey, we are going to learn about the setting a working directory, the R library (package) ecosystem, indexing, importing a spreadsheet file, and manipulating the data in the file using the *verbs* that are available in one of the libraries that we are going to import, namely dplyr.

## Setting the working directory

RStudio has a very powerful project structure. I find it easier to simply save my R markdown file (this document) and my spreadsheet file in the same directory (folder) on my computer drive.

Once I create an R markdown file, I hit the Save button (or go to File > Save on the menu bar) to save the file with an appropriate name, in an appropriate directory (folder). I then make sure that my spreadsheet file is in the same directory (folder).

We can use code to find the address on our computer drive, where the file was actually stored. The getwd() function stands for \_get working directory. Note that you have to save the R markdown file first, before calling this function. Let’s create a code chunk by hitting **CTL ALT I** ( **CMD OPT I** on a Mac). Remember to get into the habit to write a unique, short, and descriptive name to the chunk. Also leave some code comments using a hash-tag.

# Looking for the current working directory on my computer drive  
getwd()

## [1] "C:/Users/juank/OneDrive/R/My courses/Postgraduate course in medical statistics"

The directory (folder) that contains your file is shown. Note that even on a Windows computer, forward slashes are used. If you are familiar with Windows, you know that it uses backslashes to shown the directory structure.

Now, we need to tell R to actually use this directory (folder) as its active directory (folder) when searching for the spreadsheet file that we will import. This is why I save the two files together. If the spreadsheet file is in another directory (folder), you have to type the complete directory (folder) structure address to that file. That is just too much work! To set the active director (folder) to the current directory (folder), we use the setwd() function and place the getwd() as its argument.

setwd(getwd())

I usually place this right inside of the opening *setup* junk that RStudio creates as boilerplate code.

Before we get to importing and working with spreadsheet files, let’s create some simulated data so that we can look at indexing (or addressing).

## Creating computer variables to work with

First, we will create a computer variable called age to hold random value between and (inclusive). The sample() function means that we will take it from uniform distribution, where every value in the sample space of the statistical variable (i.e. ) has an equal likelihood of being selected.

set.seed(123) # So that we all get the same pseudorandom numbers  
age <- sample(15:85,  
 20,  
 replace = TRUE)

Now that we have a computer variable called age, when can introduce the length() function. It counts how many elements there are in the object that we pass as argument.

length(age)

## [1] 20

Indeed, because we asked for elements when we created our list object, we get an answer of 20.

We can also print all values to the screen by just calling the computer variable.

age

## [1] 35 70 44 77 81 18 52 78 54 47 82 47 63 55 22 78 32 17 38 82

## Using indexing (addressing) to select specific values

You will notice that the age variable appears in the Environment tab on the top right-hand side of the screen. It states int [1:20] and then starts listing the actual values. From this we can guess that every element’s address is the number through . this is indeed so. Below, the, we use square brackets (indicating indexing) and ask for the value with an index (address) of 1.

age[1]

## [1] 35

When we look at the list we printed out above, we see that was indeed the first value. Remember that this will only be true for you if you also used set.seed(123). Let’s have a look at the fifth value, which should be .

age[5]

## [1] 81

We can view a range of values by using the range symbol, :. Let’s get the first five values, i.e. 1:5.

age[1:5]

## [1] 35 70 44 77 81

If we only want values one and five, we pass these values as an atomic vector, using the c() function as address.

age[c(1, 5)]

## [1] 35 81

Now we consider something a bit more interesting. What about selecting values based on a rule? In the code chunk below we ask for only values that are equal to larger than . Note that we use the computer variable name twice!

age[age >= 50]

## [1] 70 77 81 52 78 54 82 63 55 78 82

We can specify more than one rule. Below, we ask for all values that are less than **or** more than . The symbol for or is the straight, vertical line, which is above the enter key on my keyboard.

age[age < 30 | age > 50]

## [1] 70 77 81 18 52 78 54 82 63 55 22 78 17 82

Note that every value is either below or larger than .

Good, let’s import our spreadsheet file. Wait, wait, wait, first let’s import a library.

## Importing a library

Packages or libraries are written by the huge community of developers of the R language from all over they world, kindly donating their skills and time to make our lives easier! Libraries are installed in the Packages tab on the bottom right panel. Simply type the name of the package in the search box after hitting the Install button. Allow time for the package to be installed from the internet.

After installation, we have to import the library for use in our current R markdown file. So, go ahead and first install the readr and the dplyr libraries. Below, we then import them (individually) using the library() function. Note that after the name of the chunk, is a comma and then message=FALSE and then another comma and the warning=FALSE. This can be added manually, or by deselecting the buttons after hitting the little gear icon of a chunk. The effect is that message and warning that are sometimes displayed on importing libraries are not shown. In most cases these are superfluous.

library(readr)  
library(dplyr)

The dplyr library contains all the function which we will use to extract data from our imported spreadsheet file.

## Importing a data file

The readr library contains a function called read\_csv(). There is a read.csv() function in base R (without importing the readr library). The read\_csv() function is more useful, though. It creates an object called a *tibble*. A tibble has some advantages over a traditional *dataframe*. Tibbles print nicely to most screen and it interprets categorical variables as well, categorical variables, as opposed to factors. More about this in future tutorials.

Note that the spreadsheet file is in .csv or *comma separated values* format. This is much more generic than Microsoft .xslx format. When you are in Excel, simply save the file as a .csv file. Its trips away all the bells-and-whistles added by Microsoft that really just clutter things up. If given a choice, always select **MS-DOS csv**.

In the chunk below, we import the ProjectData.csv file. Because it is in the same directory (folder) than the R markdown file, we can just refer to it, without using a long computer drive address, simply because we use the setwd(getwd()) functions above. We import the spreadsheet file in the computer variable df.

df <- read\_csv("ProjectData.csv")

## Exploring the data

Note that there is now a df variable in the Environment tab on the top right of the interface. To the far right of it is a little spreadsheet file icon. Click it to open a spreadsheet-like view of the imported data. You can also write View(df) in a code chunk.

The first thing to do is to look at the size of our data, i.e. how many rows of data (how many subjects in our study) and how many variables (columns). We use the dim(), short for dimensions, function.

dim(df)

## [1] 500 8

We can view the statistical variable names (in the columns in row 1), using the names() function.

names(df)

## [1] "Age" "Difference" "CRP" "Group" "sBP"   
## [6] "Weight" "SideEffects" "Survey"

We see a list of the eight variables. Note how every variable name is descriptive of the data that it represents. Note also that there are no *illegal* characters in the variables names, such as spaces, parentheses, and so on.

To see the values in a variable (each row in that column), we use the dollar $ notation. Let’s look at the chunk below. We do not want to list all values, so we use indexing (with square brackets remember) to show the first five rows of the *Age* statistical variable.

df$Age[1:5]

## [1] 60 65 53 57 52

What about the first five *Difference* values?

df$Difference[1:5]

## [1] -1.7 0.8 0.5 0.2 -0.8

Now the first five *Group* values.

df$Group[1:5]

## [1] "I" "I" "I" "II" "I"

Have a look at the first five values for each of the seven variables. Can you identify the variable type?

The head() function displays the first six rows of all the variables. Don’t use it too often. If you have many variables, printing to the screen will be difficult.

head(df)

## # A tibble: 6 x 8  
## Age Difference CRP Group sBP Weight SideEffects Survey  
## <dbl> <dbl> <dbl> <chr> <dbl> <dbl> <chr> <dbl>  
## 1 60 -1.7 5.8 I 104 221 No 5  
## 2 65 0.8 8.5 I 128 144 No 5  
## 3 53 0.5 8 I 128 146 No 3  
## 4 57 0.2 7.7 II 123 126 No 4  
## 5 52 -0.8 6.9 I 121 208 Yes 5  
## 6 56 0.7 7.8 I 131 168 No 4

We are not restricted to the first six rows, but can actually specify the number of rows with an argument.

head(df,  
 10)

## # A tibble: 10 x 8  
## Age Difference CRP Group sBP Weight SideEffects Survey  
## <dbl> <dbl> <dbl> <chr> <dbl> <dbl> <chr> <dbl>  
## 1 60 -1.7 5.8 I 104 221 No 5  
## 2 65 0.8 8.5 I 128 144 No 5  
## 3 53 0.5 8 I 128 146 No 3  
## 4 57 0.2 7.7 II 123 126 No 4  
## 5 52 -0.8 6.9 I 121 208 Yes 5  
## 6 56 0.7 7.8 I 131 168 No 4  
## 7 54 -0.3 7.1 II 127 210 Yes 4  
## 8 51 1.1 8.4 II 138 150 No 4  
## 9 65 -0.5 7.4 I 119 115 No 1  
## 10 60 -1.7 5.3 II 110 155 Yes 3

## The dplyr library

The dplyr library provides some of the most useful functions ever to have been added to R. They follow a convention that is slightly different from R and is referred to as the *tidyverse*, or sometimes (against his will) as the *Hadley-verse* after its inventor, Hadley Wickham.

The tidyverse set of libraries allow for the use of the pipe, %>% operator. A frightful thing when you first start using it, but an absolute powerful delight when you get used to it.

The functions in dplyr are called *verbs* and for good reason. Let’s start by looking at the filter() function. Note that when referring to dplyr I will use the terms **function** and **verb** interchangeably.

### filter

Our first function in dplyr is the filter(). As with indexing (addressing), it allows us to provide a recipe for selecting data. So, let’s look down the *Age* column and return only the values greater than . You will note that the first argument is the actual computer variable that holds our spreadsheet file. The second argument references the statistical variable (column name) and the recipe. Note that we don’t use the $ notation with these verbs.

filter(df, Age > 80)

## # A tibble: 25 x 8  
## Age Difference CRP Group sBP Weight SideEffects Survey  
## <dbl> <dbl> <dbl> <chr> <dbl> <dbl> <chr> <dbl>  
## 1 81 -0.5 6.7 II 123 217 No 2  
## 2 81 0.1 7.2 I 125 181 Yes 4  
## 3 81 -1.1 5.9 II 113 188 Yes 2  
## 4 81 -1.5 5.6 II 115 225 Yes 2  
## 5 81 -0.7 6.6 II 117 192 Yes 3  
## 6 82 0.5 8.2 I 132 179 No 1  
## 7 82 1 8.8 I 138 157 Yes 2  
## 8 82 1.9 9.8 I 140 170 No 5  
## 9 82 -0.8 6.4 II 113 186 No 3  
## 10 82 -0.6 7.4 I 116 124 Yes 2  
## # ... with 15 more rows

We see all the rows in the dataset returned, but only for those patients who are older then .

Before we carry on, let’s plunge right into using the pipe operator. The sooner we rip that plaster off, the better. Are you ready? Here comes the pain.

df %>% filter(Age > 80)

## # A tibble: 25 x 8  
## Age Difference CRP Group sBP Weight SideEffects Survey  
## <dbl> <dbl> <dbl> <chr> <dbl> <dbl> <chr> <dbl>  
## 1 81 -0.5 6.7 II 123 217 No 2  
## 2 81 0.1 7.2 I 125 181 Yes 4  
## 3 81 -1.1 5.9 II 113 188 Yes 2  
## 4 81 -1.5 5.6 II 115 225 Yes 2  
## 5 81 -0.7 6.6 II 117 192 Yes 3  
## 6 82 0.5 8.2 I 132 179 No 1  
## 7 82 1 8.8 I 138 157 Yes 2  
## 8 82 1.9 9.8 I 140 170 No 5  
## 9 82 -0.8 6.4 II 113 186 No 3  
## 10 82 -0.6 7.4 I 116 124 Yes 2  
## # ... with 15 more rows

It’s actually quite easy. We take out the first argument, which is the computer variable name and put it before the pipe. The pipe actually just say: “Take what is to the left of me and pass it as first argument to the function that is to my right. Thank you very much for your cooperation.”. By the way, the keyboard shortcut for the pipe operator is **SHFT CTRL M** ( **SHFT CMD M** on a Mac).

Let’s try that again and select only patients that have a *I* marked as their *Group* data point value.

df %>% filter(Group == "I")

## # A tibble: 251 x 8  
## Age Difference CRP Group sBP Weight SideEffects Survey  
## <dbl> <dbl> <dbl> <chr> <dbl> <dbl> <chr> <dbl>  
## 1 60 -1.7 5.8 I 104 221 No 5  
## 2 65 0.8 8.5 I 128 144 No 5  
## 3 53 0.5 8 I 128 146 No 3  
## 4 52 -0.8 6.9 I 121 208 Yes 5  
## 5 56 0.7 7.8 I 131 168 No 4  
## 6 65 -0.5 7.4 I 119 115 No 1  
## 7 54 -1.1 6.3 I 111 217 No 1  
## 8 50 -0.5 7.2 I 121 172 No 4  
## 9 52 0 7.7 I 129 148 Yes 1  
## 10 60 0.6 7.8 I 135 119 Yes 3  
## # ... with 241 more rows

We use the double equal sign, == to ask what is referred to as a Boolean question. It goes row by row and check whether the value is indeed *I*. If so, an answer of TRUE is returned and that row is included.

Now let’s get fancy and provide two recipes. Patients older than **and** who are in group I. he and symbol is &. Each recipe is placed in a set of parenthesis.

df %>% filter((Age > 80) & (Group == "I"))

## # A tibble: 12 x 8  
## Age Difference CRP Group sBP Weight SideEffects Survey  
## <dbl> <dbl> <dbl> <chr> <dbl> <dbl> <chr> <dbl>  
## 1 81 0.1 7.2 I 125 181 Yes 4  
## 2 82 0.5 8.2 I 132 179 No 1  
## 3 82 1 8.8 I 138 157 Yes 2  
## 4 82 1.9 9.8 I 140 170 No 5  
## 5 82 -0.6 7.4 I 116 124 Yes 2  
## 6 82 0 7.3 I 123 124 Yes 5  
## 7 83 -2 5.9 I 105 122 No 4  
## 8 83 -0.8 7 I 120 161 Yes 2  
## 9 84 0.5 8.5 I 129 190 Yes 4  
## 10 84 -0.4 7.4 I 120 230 Yes 4  
## 11 84 -0.6 6.9 I 120 124 No 3  
## 12 84 0.4 8.4 I 129 203 Yes 2

Great, let’s go for another verb

### select

The select() function allows us to return only the statistical variables (columns) in which we are interested. Let’s only get the *Age* variable.

df %>% select(Age)

## # A tibble: 500 x 1  
## Age  
## <dbl>  
## 1 60  
## 2 65  
## 3 53  
## 4 57  
## 5 52  
## 6 56  
## 7 54  
## 8 51  
## 9 65  
## 10 60  
## # ... with 490 more rows

Great. Now for something very fancy. It showcases the power of the pipe operator. We want only patients who are older than and who are in group I and then we only want to see the *SideEffects* and the *Survey* variables.

df %>% filter((Age > 80) & (Group == "I")) %>%   
 select(SideEffects, Survey)

## # A tibble: 12 x 2  
## SideEffects Survey  
## <chr> <dbl>  
## 1 Yes 4  
## 2 No 1  
## 3 Yes 2  
## 4 No 5  
## 5 Yes 2  
## 6 Yes 5  
## 7 No 4  
## 8 Yes 2  
## 9 Yes 4  
## 10 Yes 4  
## 11 No 3  
## 12 Yes 2

Whoa!

In the first tutorial, we created a computer variable by giving it an appropriate name and then we either manually entered some values using the c() function or we created random values using the sample() or the rnomr() functions. By using the filter() and the select() function, we can create similar computer variables to hold lists of values that we can then pass as arguments to the statistical tests that we want to conduct.

So, let’s set up a very simple research question. We want to divide our patients into those that are in group I and those that are in group II. When then want to create a list of the CRP values for each group. Let’s name the first list of CRP value group\_I\_CRP and the group II equivalent, group\_II\_CRP. We will combine our two functions, filter() and select() to do this.

group\_I\_CRP <- df %>%   
 filter(Group == "I") %>%   
 select(CRP)

Actually, we just have a new tibble with a single variable called CRP. We can convert the single variable into a vector (list of elements) using the as.vector() function and passing the tibble with the dollar sign, indicating the CRP column, as argument.

group\_I\_CRP <- as.vector(group\_I\_CRP$CRP)

Let’s do the same for group II patients.

group\_II\_CRP <- df %>%   
 filter(Group == "II") %>%   
 select(CRP)  
group\_II\_CRP <- as.vector(group\_II\_CRP$CRP)

Another way to go about this, is simply to create two tibbles. This is especially handy if we have a lot of research questions pertaining the comparing the two groups (created from the Group variable). Let’s do just this and create a group\_I and a group\_II tibble.

group\_I <- df %>%   
 filter(Group == "I")  
group\_II <- df %>%   
 filter(Group == "II")

If you now look at the Environment tab on the top right, you will see three tibbles and three computer variables that are simply vectors.

### mutate

The mutate() function allows us to create new variables (columns) in the dataset. If we look at the first six entries for the Weightcolumn, we note that the values seem to be in pounds.

head(df$Weight)

## [1] 221 144 146 126 208 168

There are pounds in each kilogram, so, we have to divide each value by . We do this through the concept of *broadcasting*. Below is a simple example of a list of five values and we simply multiply each by .

5 \* c(1, 2, 3, 4, 5)

## [1] 5 10 15 20 25

Note how the multiplication by 45$ is *broadcasted* to each element in the vector object.

Here goes the mutate() function.

df %>% mutate(Weight\_kg = Weight / 2.21)

## # A tibble: 500 x 9  
## Age Difference CRP Group sBP Weight SideEffects Survey Weight\_kg  
## <dbl> <dbl> <dbl> <chr> <dbl> <dbl> <chr> <dbl> <dbl>  
## 1 60 -1.7 5.8 I 104 221 No 5 100   
## 2 65 0.8 8.5 I 128 144 No 5 65.2  
## 3 53 0.5 8 I 128 146 No 3 66.1  
## 4 57 0.2 7.7 II 123 126 No 4 57.0  
## 5 52 -0.8 6.9 I 121 208 Yes 5 94.1  
## 6 56 0.7 7.8 I 131 168 No 4 76.0  
## 7 54 -0.3 7.1 II 127 210 Yes 4 95.0  
## 8 51 1.1 8.4 II 138 150 No 4 67.9  
## 9 65 -0.5 7.4 I 119 115 No 1 52.0  
## 10 60 -1.7 5.3 II 110 155 Yes 3 70.1  
## # ... with 490 more rows

This added the new variable, but the change is not permanent.

names(df)

## [1] "Age" "Difference" "CRP" "Group" "sBP"   
## [6] "Weight" "SideEffects" "Survey"

The easiest way to make the change permanent is to overwrite the original computer variable.

df <- df %>% mutate(Weight\_kg = Weight / 2.21)  
names(df)

## [1] "Age" "Difference" "CRP" "Group" "sBP"   
## [6] "Weight" "SideEffects" "Survey" "Weight\_kg"

## Conclusion

There are more verbs in the dplyr library. We will meet them in future tutorials.