Introduction to R - class 4 Data Wrangling 2

Tomasz Gaczorek, Wiesław Babik & Mateusz ChechetkinMarzena Marszałek [marzena.marszalek@doctoral.uj.edu.pl](mailto:marzena.marszalek@doctoral.uj.edu.pl)

2024-10-25

knitr::opts\_chunk$set(echo = T)

## Logical expressions

R as a whole computer architecture (including programming languages) is based on the **binary information**, the current either flows (TRUE) or not (FALSE), (capital letters on purpose) depending on the conditions. These logical values form a distinct type of data and can be, similarly to other types, combined into vectors using the same functions as for numbers and characters.

### **Exercise 1**

**Create a logical vector with 3 TRUE and 2 FALSE values and call it my\_logical. Note that they are internal R symbols so you should not use quotation marks ("").**

Expected result:

my\_logical <- c(TRUE, TRUE, TRUE, FALSE, FALSE)  
my\_logical

## [1] TRUE TRUE TRUE FALSE FALSE

*An advice: To make your code shorter you can use T instead of TRUE and F instead of FALSE.*

Formally, logical values correspond to 0 (for FALSE) and 1 (for TRUE) and behave like them in every mathematical operations.

### **Exercise 2**

**Calculate the sum of my\_logical.**

Expected result:

sum(my\_logical)

## [1] 3

**Curiosity**  
A sum of logical vectors can be used to check a data frame for the presence of NA cells. To do this, combine the is.na() function used on the data frame with the sum() function.

However, logical vectors have one important distinctive characteristic: they can be used for **subsetting**. To achieve this you need to provide logical vector with TRUE **for the elements you want to keep** and FALSE **for for the elements you want to discard**. The length of the logical vector used for subsetting has to be equal to the number of elements (e.g., columns) in the object we want to subset.

### **Exercise 3**

**Save 6 first rows of built-in CO2 dataset to a chosen variable. Then, using logical vectors return its 1st and 5th column.**

Expected result:

v1 <- CO2[1:6,]  
v1[, c(T, F, F, F, T)]

## Plant uptake  
## 1 Qn1 16.0  
## 2 Qn1 30.4  
## 3 Qn1 34.8  
## 4 Qn1 37.2  
## 5 Qn1 35.3  
## 6 Qn1 39.2

## Comparisons

Normally, no one creates logical vectors on their own. There are created automatically as the effect of different comparisons. The most common is testing for equality. It is done with double equal sign (==).

### **Exercise 4**

**Check whether in R: 5 equals 5.00 and whether π equals 3.14.**

Expected result:

5 == 5.00

## [1] TRUE

pi == 3.14

## [1] FALSE

The same stands for all others comparisons but the symbols are different:

• == - equal  
• != - not equal  
• > - greater than  
• >= - greater than or equal  
• < - less than  
• <= - less than or equal

**Curiosity**  
Double equal sign (==) is used for equality because single sign (=) is already in use. It serves as an alternative for the assigning arrow, however, for code purity the arrow is the one recommended.

Note that while comparing two vectors with the symbols shown above, R does not consider the action as one comparison. It rather compares them element by element, recycling shorter vector and resulting in logical vector of longer one’s length. It is the same rule as for mathematical operations on vectors.

### **Exercise 5**

**Manually create two vectors. One of the prime numbers and a second of even numbers. Both should belong to the range <0,11>. Check if they are equal.**

Expected result:

v5 <- c(2, 3, 5, 7, 11)  
v5a <- c(2, 4, 6, 8, 10)  
v5 == v5a

## [1] TRUE FALSE FALSE FALSE FALSE

To check whether vectors (as a whole) are identical use identical() function.

### **Exercise 6**

**Create two integer vectors from 1 to 10. Call them differently and compare them with identical() function. Then, change one value within first vector and repeat comparison.**

Expected result:

v6 <- c(1:10)  
v6a <- c(1:10)  
identical(v6, v6a)

## [1] TRUE

v6a[1] <- 2  
identical(v6, v6a)

## [1] FALSE

The other useful tool is the %in% operator. It provides information **whether elements of the first vector are present in the second one**. Note that it is focused on the first vector only, so there is **no recycling**. The result of the operation with %in% is the logical vector.

### **Exercise 7**

**Create two character vectors each consisting of a set of individual characters. The first should contain your name and the second one the name of other person from the group. Check how many letter your names have in common. Then, change the order of names and repeat the comparison.**

## Exclamation mark

Exclamation mark (!) works in R as the symbol of negation (=reversing the statement). Any logical vector preceded by ! will result in its reversal - FALSE changed into TRUE and vice versa.

### **Exercise 8**

**Having vector of 3 TRUE and 3 FALSE values, return its negation.**

Expected result:

v8 <- c(T, T, T, F, F, F)  
!v8

## [1] FALSE FALSE FALSE TRUE TRUE TRUE

More typically ! is used to negate comparisons. Note that the idea is the same: you negate logical vector by negating action which produces it. Remember that negated comparison should be enclosed in parentheses e.g. !(2 == 2).

### **Exercise 9**

**Create a sequence of integers from 1 to 100 in which each subsequent element is larger by 3 than the previous one. Then, create logical vector indicating which elements are larger than 50. Do not use > sign.**

Expected result:

v9 <- !(seq(1, 100, 3) < 50)  
v9

## [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  
## [13] FALSE FALSE FALSE FALSE FALSE TRUE TRUE TRUE TRUE TRUE TRUE TRUE  
## [25] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE

## Logical operators

The real power of logic in programming is provided by combining comparisons (use parentheses for clarity). There are two basic operators:

• & - **and** - condition is TRUE if both comparison are TRUE  
• | - **or** - condition is TRUE if at least one comparison is TRUE

### **Exercise 10**

**For an integer vector from 1 to 10, create logical vector indicating which element is smaller or larger than 5.**

Expected result:

v10 <- c(1:10)  
(v10 < 5) | (v10 > 5)

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

### **Exercise 11**

**For an integer vector from 1 to 10, create logical vector indicating which element is divisible by both 2 and 3.**

Expected result:

(v10 %% 2 == 0) & (v10 %% 3 == 0)

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

**Curiosity**  
R also uses double version of & and | operators. Their outcome is exactly the same and their role is just related to optimization. While using double operators, R checks first condition and if not needed, ignore the second one (e.g. in the “AND” statement if the first condition is FALSE, there is no need to check the second one as the result will always be FALSE). Note that they are often used when conditions are severly time-consuming.

## which() function

Frequently, the question is not about logical vectors themselves but rather about **which element of a vector fulfills given condition**. The answer is provided by which() function. It takes comparison as an argument and **return a vector of indexes** that can be used for subsetting.

### **Exercise 12**

**Construct a vector with first 20 integers divisible by 3. Which elements of it are larger or equal 21?**

Expected result:

v12 <- c(1:100)  
v12a <- (v12 %% 3 == 0)  
v12b <- v12[v12a]  
v12c <- v12b[1:20]  
  
# or  
v12 <- seq(from=3, by=3, length.out=20)  
  
v12d <- which(v12c >= 21)  
v12d

## [1] 7 8 9 10 11 12 13 14 15 16 17 18 19 20

### **Exercise 13**

**Having indexes of values larger or equal 21 from the previous exercise, return the values from the vector.**

Expected result:

v12c[v12d]

## [1] 21 24 27 30 33 36 39 42 45 48 51 54 57 60

## Subsetting with the logical expressions

As stated before subsetting can be done directly with logical vectors (TRUE for each kept element). In practice, it is even simpler. All you need to provide is a **condition instead of a coordinate** e.g. vector[condition] will return only the elements fulfilling given condition.

For example: There is a vector a <- c(1, 2, 3, 4). We want to subset elements of this vector that are greater then 2.  
The formula will look like this: a[a > 2].  
Then the result will be: [1] 3 4.

The logic behind is as follows:

* condition (here: [a > 2]) generates a logical vector - **positions are denoted as a series of TRUE and FALSE**. In this example the logical vector is: [1] FALSE FALSE TRUE TRUE.
* elements in this logical vector are used to **return the elements of the vector you want to subset**, for which coordinate equals TRUE, so here 2 last elements of the vector will be returned: [1] 3 4

Note that you cannot see the TRUE/FALSE vector itself but it is if fact generated and used during subsetting operation.

### **Exercise 14**

**For an integer vector from 1 to 100, return elements higher than vector’s mean.**

Expected result:

v14 <- c(1:100)  
v14[v14 > (mean(v14))]

## [1] 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69  
## [20] 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88  
## [39] 89 90 91 92 93 94 95 96 97 98 99 100

The same pattern applies for subsetting data frames.

### **Exercise 15**

**Using built-in dataset CO2, return observation for Qn2 plant.**

Epected result:

CO2[CO2$Plant == "Qn2", ]

## Plant Type Treatment conc uptake  
## 8 Qn2 Quebec nonchilled 95 13.6  
## 9 Qn2 Quebec nonchilled 175 27.3  
## 10 Qn2 Quebec nonchilled 250 37.1  
## 11 Qn2 Quebec nonchilled 350 41.8  
## 12 Qn2 Quebec nonchilled 500 40.6  
## 13 Qn2 Quebec nonchilled 675 41.4  
## 14 Qn2 Quebec nonchilled 1000 44.3

*An advice: To obtain the logical vector with one element for each row, you need to make comparison based on a column e.g.my\_data[my\_data$my\_column != 5, ] would result in the observations (including all columns) from my\_data in which the value for my\_column does not equal 5.*

### **Exercise 16**

**Using built-in dataset CO2, return observation from Mississippi chilled plant with an uptake higher than 20 ummol/m2 x s.**

Expected result:

CO2[CO2$Type == "Mississippi" & CO2$Treatment == "chilled" & CO2$uptake > 20.0, ]

## Plant Type Treatment conc uptake  
## 69 Mc1 Mississippi chilled 675 22.2  
## 70 Mc1 Mississippi chilled 1000 21.9

## dplyr

dplyr is an extremely popular R package that helps to easily manage the data frames.

### **Exercise 17**

**Install and load the dplyr package.**

### **Exercise 18**

**Upload the** [**rats.csv**](https://github.com/MarzenaMarszalek/Introduction_to_R_2024/blob/main/Intro_to_R_class_4/rats.csv) **file into an object called my\_data.**

Expected result (first 10 rows):

my\_data <- read.table("rats.csv", header = T, sep = ",")  
my\_data[1:10, ]

## Glycogen Treatment Rat Liver  
## 1 131 1 1 1  
## 2 130 1 1 1  
## 3 131 1 1 2  
## 4 125 1 1 2  
## 5 136 1 1 3  
## 6 142 1 1 3  
## 7 150 1 2 1  
## 8 148 1 2 1  
## 9 140 1 2 2  
## 10 143 1 2 2

### **Exercise 19**

**Modify my\_data by adding the column with ID at the beginning.**

Expected result (first 10 rows):

ID <- c(1:nrow(my\_data))  
my\_data <- cbind(ID, my\_data)  
my\_data[1:10, ]

## ID Glycogen Treatment Rat Liver  
## 1 1 131 1 1 1  
## 2 2 130 1 1 1  
## 3 3 131 1 1 2  
## 4 4 125 1 1 2  
## 5 5 136 1 1 3  
## 6 6 142 1 1 3  
## 7 7 150 1 2 1  
## 8 8 148 1 2 1  
## 9 9 140 1 2 2  
## 10 10 143 1 2 2

All subsequent functions come from the loaded dplyr package. Importantly, names of columns provided to dplyr functions **do not need quotation marks**.

### SORTING OBSERVATIONS

To sort the data, use arrange() function in the following manner: arrange(dataset, ordering\_column)

#### **Exercise 20**

**Obtain the observations from my\_data sorted with increasing levels of glycogen.**

Expected result (first 10 rows):

arrange(my\_data, Glycogen)[1:10, ]

## ID Glycogen Treatment Rat Liver  
## 1 4 125 1 1 2  
## 2 26 125 3 1 1  
## 3 36 127 3 2 3  
## 4 2 130 1 1 1  
## 5 1 131 1 1 1  
## 6 3 131 1 1 2  
## 7 25 134 3 1 1  
## 8 35 134 3 2 3  
## 9 29 135 3 1 3  
## 10 5 136 1 1 3

To obtain the descending order put the name of column inside desc() function.

#### **Exercise 21**

**Obtain the observations from my\_data sorted with the decreasing levels of glycogen.**

Expected result (first 10 rows):

arrange(my\_data, desc(Glycogen))[1:10,]

## ID Glycogen Treatment Rat Liver  
## 1 23 162 2 2 3  
## 2 11 160 1 2 3  
## 3 13 157 2 1 1  
## 4 20 155 2 2 1  
## 5 15 154 2 1 2  
## 6 18 153 2 1 3  
## 7 24 152 2 2 3  
## 8 19 151 2 2 1  
## 9 7 150 1 2 1  
## 10 12 150 1 2 3

You can also sort the data by **multiple columns**. To do this add an additional column in the following manner: arrange(dataset, ordering\_column1, ordering\_column2. Note that the priority of sorting is denoted by the order of function arguments.

#### **Exercise 22**

**Using function from dplyr package obtain the observations from my\_data sorted firstly by Treatment column and then by Rat column. Both in ascending order.**

Expected result (first 10 rows):

arrange(my\_data, Treatment, Rat)[1:10, ]

## ID Glycogen Treatment Rat Liver  
## 1 1 131 1 1 1  
## 2 2 130 1 1 1  
## 3 3 131 1 1 2  
## 4 4 125 1 1 2  
## 5 5 136 1 1 3  
## 6 6 142 1 1 3  
## 7 7 150 1 2 1  
## 8 8 148 1 2 1  
## 9 9 140 1 2 2  
## 10 10 143 1 2 2

### SUBSETTING COLUMNS

To select particular columns use the select() function in the following way: select(dataset, column\_name1, column\_name2). Note that all mentioned columns will be preserved and the rest will be discarded.

#### **Exercise 23**

**Obtain the Glycogen and Liver columns from my\_data.**

Expected result (first 10 rows):

select(my\_data, Glycogen, Liver)[1:10, ]

## Glycogen Liver  
## 1 131 1  
## 2 130 1  
## 3 131 2  
## 4 125 2  
## 5 136 3  
## 6 142 3  
## 7 150 1  
## 8 148 1  
## 9 140 2  
## 10 143 2

You can also use **minus preceding the column name** which means “**all except this column**”.

#### **Exercise 24**

**Obtain the ID, Glycogen, Treatment and Liver columns from my\_data. Use the minus (-) sign.**

Expected result (first 10 rows):

select(my\_data, -Rat)[1:10, ]

## ID Glycogen Treatment Liver  
## 1 1 131 1 1  
## 2 2 130 1 1  
## 3 3 131 1 2  
## 4 4 125 1 2  
## 5 5 136 1 3  
## 6 6 142 1 3  
## 7 7 150 1 1  
## 8 8 148 1 1  
## 9 9 140 1 2  
## 10 10 143 1 2

### SUBSETTING OBSERVATIONS

To subset the observations, use the filter() function in the following manner: filter(dataset, your\_logical\_condition). Note that logical conditions are always related to the values inside a given column.

#### **Exercise 25**

**Obtain the observations for Treatment equals 1.**

Expected result (first 10 rows):

filter(my\_data, Treatment == 1)[1:10, ]

## ID Glycogen Treatment Rat Liver  
## 1 1 131 1 1 1  
## 2 2 130 1 1 1  
## 3 3 131 1 1 2  
## 4 4 125 1 1 2  
## 5 5 136 1 1 3  
## 6 6 142 1 1 3  
## 7 7 150 1 2 1  
## 8 8 148 1 2 1  
## 9 9 140 1 2 2  
## 10 10 143 1 2 2

You can also combine several logical conditions by using logical operators (see above). Note, however, that for each observation under consideration it needs to result in a single TRUE or FALSE.

#### **Exercise 26**

**Obtain the observations for Treatment equals 3 with glycogen level higher than 135.**

Expected result:

filter(my\_data, Treatment == 3 & Glycogen > 135)

## ID Glycogen Treatment Rat Liver  
## 1 27 138 3 1 2  
## 2 28 138 3 1 2  
## 3 30 136 3 1 3  
## 4 31 138 3 2 1  
## 5 32 140 3 2 1  
## 6 33 139 3 2 2  
## 7 34 138 3 2 2

### MODIFYING COLUMNS

To create a new column based on the others use the mutate() function in the following manner: mutate(dataset, new\_column\_name = recipe\_for\_values). “Recipe for values” is often a **mathematical formula or simple mathematical function** based on values of other columns. Note that it is just a modification of given value for each observation separately.

#### **Exercise 27**

**Create a new column called log\_Gly that will be a natural logarithm transformation of Glycogen column. Overwrite my\_data**

Expected result (first 10 rows):

my\_data <- mutate(my\_data, log\_Gly = log(Glycogen))  
my\_data[1:10, ]

## ID Glycogen Treatment Rat Liver log\_Gly  
## 1 1 131 1 1 1 4.875197  
## 2 2 130 1 1 1 4.867534  
## 3 3 131 1 1 2 4.875197  
## 4 4 125 1 1 2 4.828314  
## 5 5 136 1 1 3 4.912655  
## 6 6 142 1 1 3 4.955827  
## 7 7 150 1 2 1 5.010635  
## 8 8 148 1 2 1 4.997212  
## 9 9 140 1 2 2 4.941642  
## 10 10 143 1 2 2 4.962845

### SUMMARY

Make a summary of your dataset with the summarise() function. The syntax is as follows: summarise(dataset, name\_of\_summary1 = recipe\_for\_value1, name\_of\_summary2 = recipe\_for\_value2,…). Recipe in that case is any aggregating function (e.g.mean() ) that will accept whole column (vector) and result in a single statistics. Note that this function will result in creation of new table with 1 row of summary statistics and as many columns as number of statistics mentioned.

#### **Exercise 28**

**Create a summary of my\_data containing mean, median, maximum value, minimum value and standard deviation of Glycogen column.**

Expected result:

summary\_glycogen <- summarise(my\_data,  
 mean = mean(Glycogen),  
 median = median(Glycogen),  
 max = max(Glycogen),  
 min = min(Glycogen),  
 st\_dev = sd(Glycogen))  
summary\_glycogen

## mean median max min st\_dev  
## 1 142.2222 141 162 125 9.754445

You can also count the number of observations corresponding to the groups within a given column e.g. check how many observations were collected for each treatment. To obtain it use the count() function by typing count(dataset, given\_column).

#### **Exercise 29**

**Use count() function to calculate how many treatments were used in a study.**

Expected result:

count(my\_data, Treatment)

## Treatment n  
## 1 1 12  
## 2 2 12  
## 3 3 12

### PIPELINE OPERATOR AND PIPELINE

As you probably noticed all dplyr functions have data as the first argument. Based on this characteristics you can create a pipeline where the next function **uses the output generated by the previous one**. In that case you should provide the dataset argument to the first function only.

Pipeline is created by connecting subsequent functions with %>% (pipe) operator. Remember to skip the data argument in all functions except for the first one e.g. select(dataset, column1, column2) %>% filter(column1 > 50). The shortcut for %>% operator is Ctrl + Shift + M (Windows) or Cmd + Shift + M (Mac).

**Curiosity** In R version 4.1.0 a new pipe operator is available |>. This operator is a part of base R so you don’t have to load additional packages (like loading dplyr to use %>%) to use it.

#### **Exercise 30**

**Using pipeline and my\_data: 1. select ID, Glycogen and Liver column. 2. obtain observations with glycogen level lower than 140. 3. sort it based on Glycogen column in descending order.**

Expected result (first 10 rows):

v30 <- select(my\_data, ID, Glycogen, Liver) %>% filter(Glycogen < 140) %>% arrange(desc(Glycogen))  
v30[1:10,]

## ID Glycogen Liver  
## 1 33 139 2  
## 2 27 138 2  
## 3 28 138 2  
## 4 31 138 1  
## 5 34 138 2  
## 6 5 136 3  
## 7 30 136 3  
## 8 29 135 3  
## 9 25 134 1  
## 10 35 134 3

### GROUPING

Performing any function over complete dataset is often not what you really want. Imagine 3 species with a trait of interest. It can be the case that the overall mean does not reflect the variability among species. To check value for each species separately you need to **group your dataset**. Do it by using group\_by() function in the following manner group\_by(dataset, column\_with\_groups). This produces a grouped dataframe and will cause all subsequent functions to **operate on each group separately**.

#### **Exercise 31**

**Using pipe, group\_by() function and my\_data data frame, create a summary table with mean and standard deviation of Glycogen column for each treatment separately.**

Expected result:

v31 <- my\_data %>% group\_by(Treatment) %>% summarise(mean = mean(Glycogen), st\_dev = sd(Glycogen))

*An advice: To perform any action on the whole dataset again, use the ungroup() function.*

#### **Exercise 32**

**Using pipe, group\_by() function and my\_data data frame, create a new column with the deviations of Glycogen values from the arithmetic mean in a given Treatment.**

Expected result (first 10 rows):

v31 <- group\_by(my\_data, Treatment) %>% mutate(std\_dev = sd(Glycogen))  
v31[1:10,]

## # A tibble: 10 × 7  
## # Groups: Treatment [1]  
## ID Glycogen Treatment Rat Liver log\_Gly std\_dev  
## <int> <int> <int> <int> <int> <dbl> <dbl>  
## 1 1 131 1 1 1 4.88 10.3  
## 2 2 130 1 1 1 4.87 10.3  
## 3 3 131 1 1 2 4.88 10.3  
## 4 4 125 1 1 2 4.83 10.3  
## 5 5 136 1 1 3 4.91 10.3  
## 6 6 142 1 1 3 4.96 10.3  
## 7 7 150 1 2 1 5.01 10.3  
## 8 8 148 1 2 1 5.00 10.3  
## 9 9 140 1 2 2 4.94 10.3  
## 10 10 143 1 2 2 4.96 10.3

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* **ADVANCED** \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

### JOINING

Imagine having two data frames corresponding to the same study system. In both of them there is a column with individual IDs. How to bind them together? Using cbind() is rather a bad idea as the order of observations can differ.

The solution is to use one of the \_join() functions.

df1 <- data.frame(  
 ID = c("ind\_1", "ind\_2", "ind\_3"),  
 V1 = c("red", "blue", "green")  
)  
  
# Tworzenie drugiej tabeli (ID i V2)  
df2 <- data.frame(  
 ID = c("ind\_2", "ind\_3", "ind\_4"),  
 V2 = c("black", "blue", "blue")  
)

data frame 1:

df1

## ID V1  
## 1 ind\_1 red  
## 2 ind\_2 blue  
## 3 ind\_3 green

data frame 2:

df2

## ID V2  
## 1 ind\_2 black  
## 2 ind\_3 blue  
## 3 ind\_4 blue

1. **left\_join() - join the values from second table (right) that correspond to observation in the first one (left). If there is no suitable value in the second table, NA is returned.**

left\_join(df1, df2, by = "ID")

## ID V1 V2  
## 1 ind\_1 red <NA>  
## 2 ind\_2 blue black  
## 3 ind\_3 green blue

1. **right\_join() - join the values from first table (left) that correspond to observation in the second one (right). If there is no suitable value in the first table, NA is returned.**

right\_join(df1, df2, by = "ID")

## ID V1 V2  
## 1 ind\_2 blue black  
## 2 ind\_3 green blue  
## 3 ind\_4 <NA> blue

1. **inner\_join() - return only that observations that have corresponding values in both tables.**

inner\_join(df1, df2, by = "ID")

## ID V1 V2  
## 1 ind\_2 blue black  
## 2 ind\_3 green blue

1. **full\_join() - join what can be joined but** keeps all observations**. In case of the lack of suitable value, returns NA.**

full\_join(df1, df2, by = "ID")

## ID V1 V2  
## 1 ind\_1 red <NA>  
## 2 ind\_2 blue black  
## 3 ind\_3 green blue  
## 4 ind\_4 <NA> blue

Each of the abovementioned functions can be used by typing: \_join(first\_table, second\_table, by = “shared\_column\_name”).

Notice that the name of shared column should be exactly the same in both tables (eg. ID).

#### **Exercise 33**

**Execute the code below to create a new data frame with ID and weight column. Join the observations from my\_data and new\_data data frames using proper \_join() function. Keep all observations from my\_data but only those from new\_data that have their counterparts in my\_data.**

new\_data <- data.frame("ID" = c(2:100),"weight" = rnorm(99,mean = 150,sd = 20))

Expected result (first 10 rows):

new\_data <- data.frame("ID" = c(2:100),"weight" = rnorm(99, mean = 150, sd = 20))  
v33 <- left\_join(my\_data, new\_data, by = "ID")  
v33[1:10, ]

## ID Glycogen Treatment Rat Liver log\_Gly weight  
## 1 1 131 1 1 1 4.875197 NA  
## 2 2 130 1 1 1 4.867534 145.2979  
## 3 3 131 1 1 2 4.875197 159.3105  
## 4 4 125 1 1 2 4.828314 168.2503  
## 5 5 136 1 1 3 4.912655 190.0491  
## 6 6 142 1 1 3 4.955827 173.1705  
## 7 7 150 1 2 1 5.010635 157.9788  
## 8 8 148 1 2 1 4.997212 147.0670  
## 9 9 140 1 2 2 4.941642 174.4356  
## 10 10 143 1 2 2 4.962845 142.6313

**Curiosity**  
1. Using joins let us avoid repeating the same information across many data frames.  
2. The concept of joins is common in many computer languages but it is most often used in databases management (e.g. SQL).

## Homework

1. Create a vector from 1 to 50. Then, using R only, calculate how many elements of this vector are higher of equal than the positive square root of 100. \*Positive square root can be calculated with the sqrt() function. 2. Upload the built-in CO2 data set as hw\_data. Using logical expressions, display the rows of the data set that have concentration below 350 AND uptake of over 30. 3. Load the dplyr package. In the hw\_data data frame sort the observations first by concentration, then by uptake. Overwrite the hw\_data file so that the columns are permanently sorted in this manner. 4. Using dplyr function add a new column to the hw\_data data frame that displays the uptake value as a percent of the maximum value of that column (=so that the highest value, 45.5, corresponds to 100%). Then remove the original uptake column using a dplyr function. 5. Use the pipe operator to perform multiple functions on the hw\_data data frame. Create a summary data frame that displays mean and standard deviation of the new uptake column for Treatment groups. Save this summary as a .csv file called “CO2\_uptake\_summary\_Your\_Name.csv”.

**Upload both your R script and .csv file to the *Pegaz* platform.**