MODULE 2: PLAYING WITH DATA!

7316 - INTRODUCTION TO DATA ANALYSIS WITH R

Mickaël Buffart (mickael.buffart@hhs.se)

TABLE OF CONTENTS

1. Data as objects

In R, you manipulate objects only. **Everything** in R is an object. An object has:

- **Attribute(s)**, *e.g.*, names, levels, etc.,
- A **type**, *e.g.*, the object is an integer, a character, etc.,
- **Value(s)**, the content of an object

To define an object, use the arrow `<-`. For example, `x <- 4` will assign 4 as the value to an object named x. The type of x will be numeric because 4 is a numeric value.

```
# Define the object named x with a numeric value of 4
x <- 4

# Define the object y, a copy of x
y <- x

# Define the object z, the result of an addition
z <- 2 + 2
```

```
# Redefine z as the square of z
z <- z^2

# Print x, y, and z
x
```

```
[1] 4
```

```
y
```

```
[1] 4
```

```
z
```

```
[1] 16
```

Note: In R, there is no distinction between defining and redefining an object (like `gen`/`replace` in Stata). You can always change an object's values, types, and attributes on the fly.

1.1 Data Types

The R objects may be of different types. There are five basic types in R. The other types are a composite combination of those.

1. **logical:** Data that should be interpreted as a logical statement, *i.e.,* `TRUE` or `FALSE`.
2. **integer:** `4L`. The `L` tells R to store the 4 as an integer.
3. **numeric:** `15.5`. Data should be interpreted as a floating number. Integers can be stored as numeric, but numeric may not be integers.
4. **complex:** `2+3i`. These are complex numbers with real and imaginary parts.
5. **character:** `"string or text"`. This is a string of characters. `"15.5"` could be stored in an object of class *character*. In that case, it would not be treated as a number but as a string of characters.

1.2 Define the Type of an Object

In R, you do not need to state the type of an object before assigning a value to it (as is the case in C language, for example). Types are defined on the fly. R can also convert some objects to other types on the fly. For example:

```
a <- 2L # a is an integer

b <- 4.4 # b is a numeric

a <- a + b # a is now a numeric
```

You will get an error if R cannot convert the object to the desired type during an operation.

```
a <- 2L # a is an integer

b <- "four" # b is a character

# a + b generates an error because b cannot be converted into a numeric or integer
and characters cannot be added to an integer
a + b
```

Sometimes, you need to redefine the type of an object manually. For example, you want to coerce a character object (`a <- "2"`) into an integer (`a <- 2L`). Common commands can help you with this when the data is formatted suitably:

- `as.integer()` will take data that *look like integers* but is formatted as another type and change it to an `integer`.
- `as.numeric()` does the same but change it to `numeric` type.
- `as.character()` coerce an object into an object of type `character`.
- other `as.something()` functions exist. You will discover them as you progress with R.

1.3 Assess the Type of an Object

In some logical operations, you may need to check if an object is of the desired type. You can perform this with the command `is.integer()`, `is.logical()`, `is.character()`, etc. Depending on the object type, this function will return a logical value (TRUE or FALSE).

```
a <- 2L
is.integer(a)

b <- 2
is.integer(b)
```

Question: In the example above, what will be the outcome of `is.integer(b)`?

1.4 Mathematical operations on R objects

As we have seen with the examples above, applying mathematical operations on R objects defined above is possible. These include additions, subtractions, multiplication and division, exponentiation, logarithms, or any other mathematical function that has a definition in R.

```
# Addition and Subtraction
2 + 2 - 3
```

```
[1] 1
```

```
# Multiplication and Division
2 * 2 + 2 / 2
```

```
[1] 5
```

```
# Exponentiation and Logarithms
2^2 + log(2)
```

```
[1] 4.693147
```

1.5 Logical operations on R objects

You can also evaluate logical expressions in R:

```
# Less than
a < b
```

```
[1] FALSE
```

```
# Less than or equal
a <= b
```

```
[1] FALSE
```

```
# More than
a > b
```

```
[1] TRUE
```

```
# Greater than or equal
a >= b
```

```
[1] TRUE
```

```
# Equals
a == b
```

```
[1] FALSE
```

```
# Not equals
a != b
```

```
[1] TRUE
```

```
# Another negation
!TRUE == FALSE
```

```
[1] TRUE
```

You can also use `&` (`AND`) and `|` (`OR`) operations with logical expressions:

```
a <- 12
b <- 15
c <- a

# Is 5 equal to 5 OR is 5 equal to 6?
(a == b) | (a == c)
```

```
[1] TRUE
```

```
# Is 5 equal to 5 AND is 5 equal to 6?
(a == b) & (a == c)
```

```
[1] FALSE
```

2. Data structures

So far, we considered only objects containing a single value. When you do data analytics, you often manipulate objects containing multiple values, such as **vectors**, **lists**, **matrices**, or **dataframes** (*i.e.,* tables). In R, these objects can all coexist in the same environment.

2.1 Vectors

The basic data structure containing multiple elements in R is the **vector**.

- An R vector is much like the typical view of a vector in mathematics, *i.e.,* it is a 1D array of elements.
- Usually, when we talk about *vectors*, we mean **atomic vectors**, the typical type of vectors in R, that are of a single *type*. The types of vectors are the same as the single value object we described before (in practice, the objects presented above are atomic vectors of length 1).

- Contrary to other languages, like C++, vectors in R are dynamic: they have the ability to resize automatically when you add or delete an element. You can assess the length of a vector with the function `length(vector_name)`.

2.1.1 Create Vectors

- To create a vector, use the coerce function: `c()`.

```
# Create a `days` vector
days <- c("Mon", "Tues", "Wed", "Thurs", "Fri")

# Create a `numbers` vector
numbers <- c(13, 18, 17, 20, 21)

# You could as well coerce multiple single-value objects into a vector
a <- 12
b <- 15
c <- a

d <- c(a, b, c)

# Display the vector d
d
```

```
[1] 12 15 12
```

- Because atomic vectors are of a single type, assigning values of different types to a vector will coerce them to the most general type.

```
# Create a vector with values of different types
vector_lambda <- c("Mon", TRUE, 2L)
```

Question: What will be the type of `vector_lambda`?

2.1.2 Name Vectors

You can name the element of a vector by assigning a vector of names to your vector.

```
# Create a vector
vec_a <- c("a", "b", "c", "d")

# Naming elements in vec_a
names(vec_a) <- c("a first name", "another name",
                  "yet another name", "the 5th element")

# Display `vec_a` vector
vec_a
```

```
    a first name     another name yet another name  the 5th element
             "a"              "b"              "c"              "d"
```

```
# Display the names of the elements in vec_a
names(vec_a)
```

```
[1] "a first name"     "another name"     "yet another name" "the 5th element"
```

Warning: In the example above, the element names in the vector are arbitrary. For example, element 4 in the vector is named the 5th element. You decide how you name your elements.

2.1.3 Subset Vectors from Element Position

- If you would like to call a specific element of a vector into another object (it is called subsetting), you need to write the position of this element into brackets `[ ]`.

```
# extract the third element of vec_a into vec_b
vec_b <- vec_a[3]

# display vec_b
vec_b
```

```
yet another name
             "c"
```

Notes: As we have seen before, a vector may contain only one element. You will receive an error if you invoke an element that does not exist.

Notes: In R, the positions in a vector are numbered from 1 to n, n being the length of the vector (while in Python, they are numbered from 0 to n-1).

- You may also call multiple elements of a vector, by calling all of the positions of interest at the same time

```
# A vector
a <- c(1, 15, 654, 8, 5, 891, 1, 2, 34, 9, 7)

# Call the elements 2, 5 and 7
a[c(2, 5, 7)]
```

```
[1] 15  5  1
```

- You can also call all the elements between two bounds, with the operation `:`

```
# Call all the elements of a between the 2nd and the 6th element
a[2:6]
```

```
[1]  15 654   8   5 891
```

- If the vector elements are named, it is possible to call an element by its name, with quotation marks.

```
# Display the third element of vec_a, while calling it by its name
vec_a["yet another name"]
```

```
yet another name
             "c"
```

```
# Call the same element by its position in the vector
vec_a[3]
```

```
yet another name
             "c"
```

2.1.4 Subset Vectors with Conditions on Values

- You can also call elements of a vector based on a value condition

```
vec_c <- c(2, 5, 8, 18, 65, 1, 23, 45)

vec_c[vec_c >= 18]
```

```
[1] 18 65 23 45
```

Note: The position of the elements in the vector after subsetting has changed, because some elements have been removed (in the example, 1, 2, 5, and 8 are under 18, so they are removed).

- In addition to the logical operations mentioned earlier, vectors can be subsetted with the operator `%in%`. Identifying elements of a vector belonging to a specific ensemble is very convenient.

```
vec_c <- c(2, 5, 8, 18, 65, 1, 23, 45)

vec_subensemble <- c(2, 8, 32)

# Extract all the elements of vec_c that are in vec_subensemble
vec_c[vec_c %in% vec_subensemble]
```

```
[1] 2 8
```

2.1.5 Replace a Value in a Vector

You can assign a new value to an element of a vector just like you do it for any object.

```
# Replacing the 3rd element of vec_a with value 84.
a[3] <- 84

# That also works with other forms of calling.
# For example, replace elements 5, 6, and 7 with 444
a[5:7] <- 444

# Display the new a
a
```

```
 [1]   1  15  84   8 444 444 444   2  34   9   7
```

```
# Replace all the element of vec_c that are in vec_subensemble
vec_c[vec_c %in% vec_subensemble] <- 65
```

question: What will be the content of `vec_c`?

2.1.6 Change and Assess a Vector Type

Because *atomic* vectors are of a single type, it is possible to use the commands `as.integer()`, `as.character()`, etc., to change the type of all the vector elements. If it is impossible to coerce an element of the vector to the new type, it will be replaced by a missing value, `NA`.

```
vec_d <- c("2", "5", "six", "18")
```

Question: What would be the output of `as.integer(vec_d)`?

2.1.7 Composite Types

2.1.7.1 Factors

In the first part of this module, we have seen 5 types of objects (integer, numeric, character, logical, and complex). The five types are the bricks to build any other data types.

For example, R is known to have a **factor** type used for categorical variables. The *factor* is an integer, where each single value is associated with a string named level.

```
# Creating a factor
factor_vector <- factor("blue", "red")

# The class of factor_vector is "factor", but the type is an integer, with 2 level
s: blue and red.
class(factor_vector)
```

```
[1] "factor"
```

```
typeof(factor_vector)
```

```
[1] "integer"
```

- It is then possible to extract the list of levels with `levels(object)`.

```
levels(factor_vector)
```

```
[1] "red"
```

- With `as.factor()`, you can reformat a character vector into a factor object. Each unique element of the vector will be used as a level of the factor, in alphabetical order.

```
# a character vector
color <- c("blue", "red")

# A factor vector
color_fac <- as.factor(color)
```

- The alphabetical order of the levels is not always the best suitable. For example, when you describe a Likert-scale, you want the levels to go from *strongly disagree* to *strongly agree*, not from *neither agree nor disagree* to *strongly disagree*... In that case, you can reorder the factor levels, with the `factor()` function.

```
# A Likert scale as a character vector
lik <- c("Strongly disagree", "Somewhat agree", "Strongly agree", "Neither agree n
or disagree", "Strongly disagree", "Somewhat agree", "Somewhat disagree")

# Converted as factor
test_1 <- as.factor(lik)
levels(test_1)

# Changing order of the levels
test_2 <- factor(lik,
                 levels  = c("Strongly disagree", "Somewhat disagree",
                             "Neither agree nor disagree",
                             "Somewhat agree", "Strongly agree"),
                 ordered = TRUE)

levels(test_2)
```

Question: What is the difference between `test_1` and `test_2`?

2.1.7.2 Dates

Another composite type, **Date**, is, from the R interpreter perspective, an integer counting the number of days that occurred since the first of January 1970. It is important to understand how data are stored and manipulated by the R interpreter to use and transform them efficiently.

```
# A date, in a character vector
d <- "1970-11-05"

# Convert d to a date composite type
d <- as.Date(d)

# Convert the new d (a date), to an integer
d_int <- as.integer(d)

# Display d: 308 days between 1970-01-01 and 1970-11-05
d_int
```

```
[1] 308
```

- This means you can perform mathematical operations on a date as you would on an integer.

2.1.8 Operations on Vectors

Operations on vectors are element-wise. So if 2 vectors are added together, each element of the 2^{nd} vector would be added to the corresponding element from the 1^{st} vector. Operations can be performed on vectors of different lengths if the length of one is a multiplier of the length of the other

```
vec_f <- c(8, 10, 10, 15, 16)
vec_g <- c(18, 3, 1, 10, 5)
vec_h <- c(18, 3, 1, 10)

vec_f * 2


vec_f + vec_g


vec_f + vec_h
```

Question: What will be the output of the example above?

- An example using dates. Imagine you would like to know the average frequency of a client's visit to your shop.

```
# Client visit
d <- c("2023-01-12", "2023-01-19", "2023-02-24",
       "2023-03-13", "2023-04-08", "2023-04-10",
       "2023-06-17", "2023-07-03")

# Convert the character into dates
d <- as.Date(d)

# Now, d is an integer... Counting numbers of days between visits
next_visit <- d[2:length(d)]
previous_visit <- d[1:length(d)-1]

# days between visits
days_visits <- next_visit - previous_visit

# average
mean(days_visits)
```

```
Time difference of 24.57143 days
```

Note:

- `2:length(d)` means *all the elements from the* 2^{nd} *element until the length of d.*
- `1:length(d)-1` means *all the elements from the first element until the length of d - 1* (i.e., all the elements except the last one).
- We need to exclude the last elements from d because we can only subtract elements from vectors of the same length

- The subtraction `next_visit - previous_visit` will subtract all the pair of elements in the vectors, as would be the case in a subtraction of vectors in maths.
- `mean()` is a function that computes the average. We will study functions later in this course.

2.2 Lists

A **list** is a vector that can store elements of different types and lengths. Lists are a common format when you collect nested data from the web (for example, the json output of the Twitter API is a list).

2.2.1 Create a List

You create a list with the function `list()`.

```
# Some vectors
vec_f <- c(8, 10, 10, 15, 16)
vec_i <- c("en", "ett", "yes, I am a beginner in Swedish")
vec_j <- c(TRUE, TRUE, FALSE, FALSE, FALSE, FALSE, FALSE, TRUE)

# Create a list
list_a <- list(vec_f, vec_i, vec_j)

# Display the list
list_a
```

```
[[1]]
[1]  8 10 10 15 16

[[2]]
[1] "en"                                "ett"
[3] "yes, I am a beginner in Swedish"

[[3]]
[1]  TRUE  TRUE FALSE FALSE FALSE FALSE FALSE  TRUE
```

As for vectors, it is possible to name elements of a list. For example:

```
# Create a list
list_a <- list(obs      = vec_f,
               text     = vec_i,
               logi_var = vec_j)

# Display the list
list_a
```

```
$obs
[1]  8 10 10 15 16

$text
[1] "en"                                "ett"
[3] "yes, I am a beginner in Swedish"

$logi_var
[1]  TRUE  TRUE FALSE FALSE FALSE FALSE FALSE  TRUE
```

In the example above, the first element of the list is named `vec_f` and is given the elements of `vec_f` as value. Note that `vec_f` within the list and `vec_f` outside the list are two different and independent objects.

2.2.2 Extract Elements from a List

With double brackets `[[ ]]`, you can extract elements from a list:

```
# Extract the second element of the list
list_a[[2]]
```

```
[1] "en"                                "ett"
[3] "yes, I am a beginner in Swedish"
```

```
# Or extract it using its name
list_a[["vec_i"]]
```

```
NULL
```

You can also extract a specific value from a specific element of a list:

```
# Extract the third value of the second element of the list
list_a[[2]][3]
```

- The position of the list element is stated first, with double brackets, the position of the value to extract within the list element is stated second, with single brackets, because the list simply contains a series of atomic vectors (this also means that elements of the list can be stored as atomic vectors[1]).
- If the value to extract does not exist, `NA` is returned.

2.3 Matrices

- Data in a 2-dimensional structure can be represented in two formats: a `matrix` or a `dataframe`.
- A matrix is used for 2D data structures of a single data type (like atomic vectors). Usually, matrices are composed of numeric objects.
- To create a matrix, use the `matrix()` command. The syntax of `matrix()` is:

```
matrix(x,
       nrow  = a,
       ncol  = b,
       byrow = FALSE/TRUE)
```

- `x` is the data that will populate the matrix.
- `nrow` and `ncol` specify the number of rows and columns, respectively. Generally need to specify just 1 since the number of elements and a single condition will determine the other.
- `byrow` specifies whether to fill in the elements by row or column. The default is `byrow = FALSE`, *i.e.,* the data is filled in by column.

2.3.1 Create a matrix from scratch

A simple example of creating a matrix would be:

---

[1] In fact, it is possible to store more complex objects than atomic vectors as element of a list, but that would be undoubtedly going a bit too far.

```r
vec_k <- c(18, 21, 31,
           10,  8,  6)

matrix(vec_k,
       nrow  = 2,
       ncol  = 3,
       byrow = FALSE)
```

```
     [,1] [,2] [,3]
[1,]   18   31    8
[2,]   21   10    6
```

Note the difference in appearance if we instead `byrow = TRUE`

```r
matrix(vec_k,
       nrow  = 2,
       ncol  = 3,
       byrow = TRUE)
```

```
     [,1] [,2] [,3]
[1,]   18   21   31
[2,]   10    8    6
```

- Note that the line breaks and spaces, when defining `vec_k`, are purely for readability. Unlike Stata, R allows you to break code over multiple lines without any extra line break syntax.
- As for vectors and lists, it is possible to name matrix elements. Only here, instead of using `names()`, we use `rownames()` and `colnames()`.

```r
m <- matrix(vec_k,
            nrow  = 2,
            ncol  = 3,
            byrow = TRUE)

rownames(m) <- c("1st row", "2nd row")

# Diplay m
m
```

```
        [,1] [,2] [,3]
1st row   18   21   31
2nd row   10    8    6
```

2.3.2 Matrix operations

- In R, matrix multiplication is denoted by `%*%`, as in `A %*% B`
- `A * B` instead performs *element-wise* (Hadamard) multiplication of matrices, so that `A * B` has the entries a_1b_1, a_2b_2 etc.
- An important thing to be aware of with R's `A * B` notation, however, is that if either of the terms is a 2D vector, the terms of this vector will be distributed element-wise to each column of the matrix.

```r
# Create a vector
vec_l <- c(1, 2)

# Display vec_l
vec_l
```

```
[1] 1 2
```

```
# Element-wise operations with a vec_l and m, defined above
vec_l * m
```

```
        [,1] [,2] [,3]
1st row   18   21   31
2nd row   20   16   12
```

- You can transpose a matrix with:

```
# Transpose m
m2 <- t(m)

# Display m2
m2
```

```
     1st row 2nd row
[1,]      18      10
[2,]      21       8
[3,]      31       6
```

2.4 Dataframes

Dataframes are probably the most common structure you will use in R. Dataframes are generic objects in R to store tabular data. They are composed of a series of vectors of the *same length* and possibly *different types* (to the contrary of matrices that have only one type).

- Each vector is stored as a column of the dataframe.
- Each column has a name. If no names are given to a column when it is set, R will provide a default name (usually `V1` or `X1`).
- Usually, when you import data in R from a data file (such as a `.dta` Stata file, or a `.csv` file), the content will be stored in a dataframe. You can also create dataframes from vectors or matrices, manually.
- **Note:** some packages create an object called *Tibble*. The Tibble is an alternative to the `data.frame` created as a part of the `tidyverse` package. As `data.frame`, `tibble` contain a table of data, *e.g.* variables as vectors in columns, possibly of different types, and rows as values. In most cases, you would not see the difference between a `data.frame` and a `tibble`, but some functions in R require the use of `tibble`. You can simply convert `data.frame` into `tibble` with the `as.tibble()` function.

2.4.1 Create a data frame

- You generally create dataframes by loading data into R, from a file. For example:

```
# Dataset in SPSS format loaded into R
pisa_spss <- rio::import("data/CY07_MSU_SCH_QQQ.sav")
```

- Another way to create a data frame is to combine other vectors or matrices (of the same length).

```
# Create vectors
vec_a <- c("a", "b")

vec_b <- c(844, 327)
```

```r
# Structure the vectors in a dataframe
df <- data.frame(vec_a, vec_b)

# Display the dataframe
df
```

```
  vec_a vec_b
1     a   844
2     b   327
```

- Note that your dataframe, `df`, now appears in your environment tab. At the end of the line, you see a small table. If you click on it, you will view the content of the dataframe.

2.4.2 Rename Columns of `data.frame`

- You can rename the dataframe's columns with the `names()` function, as before.

```r
# Name the columns of df
names(df) <- c("column 1", "column 2")
```

- You can also rename a column using a condition.

```r
# Rename the column of df named "column 1" into "COLUMN A"
names(df)[names(df) == "column 1"] <- "COLUMN A"
```

- Or you can name the column while creating your dataframe.

```r
# Create a new data.frame with 3 columns of 5 observations: id, value, and var_3
df_2 <- data.frame(id = c("b", "a", "c", "a", "b"),
                   value = c(15, 32, 81, 654, 11),
                   var_3 = c(TRUE, FALSE, FALSE, TRUE, TRUE))
```

2.5 Subset Variables from a `data.frame`

Just like we did from vectors, you can subset elements of a dataframe with element positions and with values. Simply keep in mind that dataframes have two dimensions:

- the first dimension is the rows
- the second dimension is the columns of your dataframe

```r
# For example, call the 3rd element of the 2nd column in df_2
df_2[3, 2]
```

```
[1] 81
```

- To extract all the values of the 3rd row, write:

```r
# Extract all the values of the third row
df_2[3,]
```

```
  id value var_3
3  c    81 FALSE
```

- To extract all the values of the 3rd columns, write:

```r
# Extract all the values of the 2nd column
df_2[,2]
```

```
[1]  15  32  81 654  11
```

- As for vectors, you can extract elements (lines or columns of your tables) using row names, column names, or conditions.

```
# Extract one column by name
CNT <- pisa_spss[,"CNT"]
```

Note: To simplify the writing, when you call a column of a dataframe, instead of writing the bracket and comma, you can replace the bracket part with a dollar symbol `$`. It does the same, but the writing is simpler for the general case.

```
# Extract one column by name
CNT <- pisa_spss$CNT
```

Note that `CNT` and `pisa_spss$CNT` are two different objects. The first one is a vector in your environment. The second one is a column of the data.frame `pisa_spss`.

You can also extract multiple columns at a time `"CNT"`, `"CNTSCHID"`, `"SC016Q01TA"`, `"SC155Q02HA"`, from `pisa_spss`. In that case, you cannot use the dollar symbol.

```
# Using brackets
df <- pisa_spss[, names(pisa_spss) %in% c("CNT",
                                          "CNTSCHID",
                                          "SC016Q01TA",
                                          "SC155Q02HA")]
```

- You can also extract the first 5000 observations, or a random sample, or observations meeting a condition.

```
# Extract the first 5000 observations
df_3 <- df[1:5000,]

# Extract a random sample of 5000 observations
df_4 <- df[sample(nrow(df), 5000), ]

# Extract rows when SC016Q01TA <= 95
df_5 <- df[df$SC016Q01TA <= 95, ]

# You can also combine multiple conditions
df_6 <- df[(df$SC155Q02HA > 90) & (df$SC016Q01TA <= 95),]
```

Note: the conditional subsetting keeps the missing values.

2.6 Remove `data.frame` or Variables

You created many `data.frame` in your environment. If you want to remove some of them and save memory, you can do so with:

```
# remove pisa_spss and df_2
rm(pisa_spss, df_2)
```

The command above removes two complete datasets from your environment. If you want to remove a variable in a dataset, for example, `"CNTSCHID"`, use the following command:

```
# Remove columns CNTSCHID in dataframe df
df[, "CNTSCHID"] <- NULL
```

2.7 Add Observations in a `data.frame`: Append Rows

Imagine that you collected data into two different Excel files. They contain the same variables, but different observations. You can append rows to a single dataset with `plyr::rbind.fill()`. Make

sure that the columns in common have the same names. The missing columns will be filled with `NA` (missing values).

```
# Examples of Excel files to combine
f1 <- rio::import("./data/dc_1.xlsx")
f2 <- rio::import("./data/dc_2.xlsx")

# Make sure the same column have the same name in both dataframe
#  In our example, the observation identifier is called different accross
#   files. We rename it.
names(f1)[names(f1) == "identifier"] <- "id"

# The, you can append the files with plyr::rbind.fill()
df <- plyr::rbind.fill(f1, f2)
```

Now, df contains all the observations of f1 and f2.

2.8 Add Variables in a `data.frame`: New Columns

Creating new variables that are functions of existing variables in a data set can be done with:

```
# var_3 is 2 * var_2
df$var_3 <- df$var_2 * 2

# We can also write
df[, "var_3"] <- df[, "var_2"] * 2

# Recode a dummy variable from the gender variable
df$is_female <- as.integer(df$gender == "female")
```

In the example above, you apply an operation to all the values of a vector. Sometimes, you may want to create a variable whose value is conditional to another variable. For example, you want to create a `var_4` variable that is `var_2 + 1` if `var_1 >= 4` and `var_2 - 1` otherwise. You can do so by writing twice the condition: once on the output variable, and once on the operation. You have to write the condition twice to ensure that each vector element correctly matches each other: observation 1 in output matches with observation 1 in input (*REMEMBER*: when you subset a vector with a condition, elements that do not meet the condition are removed).

```
# Example: var_4 = var_2 + 1 if var_1 >= 4 and var_2 - 1 otherwise

# create a var_4 variable with the case otherwise
df$var_4 <- df$var_2 - 1

# Adjust if var_1 >= 4: the condition appears twice
df$var_4[df$var_1 >= 4] <- df$var_2[df$var_1 >= 4] + 1
```

2.9 Identify Missing Values

Does one of your variables contain missing values? You can assess if an observation is missing with the function `is.na()`. Missing values in R are coded `NA`. For example:

```
# Count the number of missing values in the gender variables
sum(is.na(df$gender))
```

```
[1] 2
```

To know the number of cases are found in a categorical or a logical variable, use `table`. The table commands counts all the occurrences of each values of a variable. Example:

```
table(is.na(df$gender))

FALSE  TRUE
   24     2
```

2.9.1 Remove Missing Values

You can remove the missing values using:

```
# Keep only the complete cases in your data.frame
df <- df[complete.cases(df), ]
```

The command above remove all the rows containing missing values on any variables of `df`.

Question: Using `is.na()`, how could you remove rows containing missing values on `gender`?

Solution:

```
# Remove rows containing missing value on the gender variable
df_2 <- df[!is.na(df$gender), ]
```

- `!is.na()` means NOT is NA.

2.9.2 Recode Missing Values

Another problem characteristic of observational data is missing data. In R, the way to represent missing data is with the value `NA`. You can recode missing values that *should be* `NA` but are coded using a different schema by using brackets:

```
# Replace 99-denoted missing data with NA
df[df$var_2 == 99, ] <- NA
```

Note: R does not naturally support multiple types of missingness like other languages, although it's possible to use the `sjmisc` package to do this.

2.10 Merge Dataframes Together

We learned how to add a vector to a `data.frame`. By doing this, we assume that:

1. the vector is of the same size as the `data.frame`, and
2. the observations are in the same order in the vector and the `data.frame`.

Let's assume this is not the case. For example, you have the two following `data.frame`, with a unique identifier and a variable, and you want to `merge` them:

```
# Create two data.frame
df_1 <- data.frame(id     = c(1, 2, 3, 4, 5, 6, 7),
                   var_1 = c("a", "a", "b", "a", "d", "e", "e"))

df_2 <- data.frame(identifier = c(2, 12, 7, 8, 9, 10, 11, 3, 13),
                   var_2      = c(TRUE, TRUE, TRUE, FALSE, FALSE, TRUE,
                                  FALSE, TRUE, FALSE))
```

You note that the two `data.frame` contain a unique identifier (named differently) and a variable each, but they do not have the same number of rows, and the unique identifiers do not match line per line.

The starting point for any merge is to enumerate the column or columns that uniquely identify observations in the dataset. Rows often have unique identifiers, such as respondent id, organization number, municipality, *etc.* For panel data, this is typically the group identifier and a time variable, for example, Sweden in 2015 in a cross-country analysis.

In the example above, the variable names of the unique identifier do not match across `data.frame`. It does not matter: with the merge function, you can provide the name of the identifier for each `data.frame`. It also does not matter if it is the first column of the `data.frame` or not.

2.10.1 Case 1: left join

You left join when you want all the observations from the first dataset, but not the second (missing values will be filled with `NA`:

```
df_left <- merge(x = df_1, y = df_2,
                 by.x = "id", by.y = "identifier",
                 all.x = TRUE, all.y = FALSE)
```

- where `x` and `y` are the names of the `data.frames` to merge.
- `by.x` and `by.y` provide the names of the unique identifiers in the first and in the second `data.frame`, respectively.
- `all.x = TRUE` indicates that we want to keep ALL the observations from the first `data.frame` (even if there is no corresponding observations in the second `data.frame`)
- `all.y = FALSE` indicates that we do NOT want to keep the observations from the second `data.frame` that do not also appear in the first `data.frame`

2.10.2 Case 2: right join

It is the same, but keeping all observations from the second dataset:

```
df_right <- merge(x = df_1, y = df_2,
                  by.x = "id", by.y = "identifier",
                  all.x = FALSE, all.y = TRUE)
```

- Pay attention to the different number of observations in `df_1` and `df_2`.

2.10.3 Case 3 and 4: all observations, or only observations in common

- To keep all observations from any data.frame (most inclusive):

```
df_all <- merge(x = df_1, y = df_2,
                by.x = "id", by.y = "identifier",
                all.x = TRUE, all.y = TRUE)
```

- To keep only the common observations (most exclusive):

```
df_all <- merge(x = df_1, y = df_2,
                by.x = "id", by.y = "identifier",
                all.x = FALSE, all.y = FALSE)
```

2.11 Extract missing observations

Sometimes, you may want to keep only observations that do NOT appear in both datasets. You can do this with R base:

```
# df_1 observations that are NOT in df_2
df1_not_df2 <- df_1[!c(df_1$id %in% df_2$identifier),]

# df_2 observations that are NOT in df_1
df2_not_df1 <- df_2[!c(df_2$identifier %in% df_1$id),]
```

Reminder: `!` means NOT. The way to read the content of the brackets `!c(df_1$id %in% df_2$identifier)` means NOT `df_1$id` in `df_2$identifier`.

2.12 Sort `data.frame`

If you want to sort your data by the values of a particular variable, use the `sort()` function:

```
# Order by increasing experience and decreasing wage
wages[order(wages[, "exper"], -wages[, "wage"]), ]
```

Note the `-` in front of the `wage` variable, to signal decreasing order.

2.13 Transforming from wide to long format: Gather

If values for a single variable are spread across multiple columns (*e.g.*, income for different years), gather, from the `tidyr` package, moves this into single "values" column with a "key" column to identify what the different columns differentiated. In short, `gather` converts a wide dataset into a long one. Example:

```
library(dplyr)
```

```
Warning: package 'dplyr' was built under R version 4.3.1
```

```

Attaching package: 'dplyr'
```

```
The following objects are masked from 'package:stats':
```

```

    filter, lag
```

```
The following objects are masked from 'package:base':
```

```

    intersect, setdiff, setequal, union
```

```
# We create a dataframe for the example
earnings_panel <- data.frame(person = c("Elsa", "Mickey", "Ariel", "Gaston",
                                        "Jasmine", "Peter"),
                             y1999 = c(10, 20, 17, 19, 32, 22),
                             y2000 = c(15, 28, 21, 19, 35, 29))

# Gather data
earnings_gathered <- earnings_panel %>%
  tidyr::gather(key = "year",
                value = "wage",
                y1999:y2000)
```

You could also do this without the tidyverse, but that would require more lines of code.

2.13.1 A parenthesis about pipes, denoted `%>%`

A famous function from the tidyverse is the ***pipe***, denoted `%>%`:

- Pipes allow you to combine multiple steps into a single piece of code.
- Specifically, after performing a function in one step, a pipe takes the data generated from the first step and uses it as the data input for a second step. Example:

```
library(dplyr)

earnings_gathered <- earnings_panel %>%
  tidyr::gather(key = "year",
                value = "wage",
                y1999:y2000)
```

This is equivalent to writing:

```
earnings_gathered <- tidyr::gather(earnings_panel,
                                   key = "year",
                                   value = "wage",
                                   y1999:y2000)
```

But the code below does not require an extra package (`dplyr`), to run... In general, I find that the pipes `%>%` makes the code less intuitive and more prone to errors. It also does not save the intermediary steps. Thus, I don't use it. However, it is often seen in books and example codes that you can find online, so you need to know what it means. You can also use it as you like.

2.14 Transforming from long to wide format: Spread

Spread tackles the other major problem, that often times (particularly in longitudinal data) many variables are condensed into just a "key" (or indicator) column and a value column. In short, spread converts a long dataset into a wide one. Example:

```
earnings_gathered %>% tidyr::spread(key = "year",
                                    value = "wage")
```

2.15 Aggregate data

Creating summary statistics by group is another routine task. You can do so with `aggregate()`. For example, let's calculate the mean wage per gender in the `Wages1` dataset:

```
# Loading data from Ecdat, see Module 1.
wages <- Ecdat::Wages1

# With base R:
aggregate(wage ~ sex, wages, mean)
```

```
     sex     wage
1 female 5.146924
2   male 6.313021
```

Note the `~` to separate `wage` and `sex`. This means *by* and is very much used in the formulas of any statistical models, as we will see later in this course.