7316 - INTRODUCTION TO DATA ANALYSIS WITH R

MODULE 2: PLAYING WITH DATA

Mickaël Buffart (mickael.buffart@hhs.se)

Table of contents

1. MODULE 2: Playing with data!

1.1 Data as objects

In R, you manipulate objects only. To define an object, use the arrow `<-`. For example `x <- 2 + 2` will assign 4 as the value of x.

Note: In R, there is no distinction between defining and redefining an object (*à la* gen/replace in Stata).

```
# Define y
y <- 4

# Redefine y
y <- y^2

# Print y
y
[1] 16
```

1.1.1 Data types

In R, you manipulate objects. Everything in R is an object. The objects may be of different types. There are 5 basic types in R[1]. 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 number with real and imaginary part.
5. character: `"string or text"`. This is a string of character. 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.

The five types above are the bricks to build any other data types. For example, R is known to have a factor type, that is used for categorical variables. From an object perspective, a factor is a specific object, but from the R interpreter perspective, the factor type is in fact an integer where each integer is assigned a string of character. Another type, Date, is in fact, from the R interpreter perspective, an integer counting the number of days that occurred since the first of January 1970. This is important to understand how data are stored and manipulated by the R interpreter to be able to use and transform them efficiently.

[1] There is in fact a sixth type, the raw data type, but its use is beyond the scope of this course.

1.1.2 Discovering the type of an object

- To discover the "type" of an object, from an object oriented perspective, you can use the function `class(object)`.
- To discover the "type" of the object, from R's point of view, you can use the function `typeof(object)`.

```
a <- 2L
class(a)
typeof(a)

b <- TRUE
class(b)
typeof(b)

c <- "True"
class(c)
typeof(c)

d <- factor("True")
class(d)
typeof(d)
```

Question: Can you guess what will be the type of `d`?

Note: In the example above, `class()` and `typeof()` return the same information for `a`, `b`, and `c` because they are basic R types. For d, because `factor` is a composite type, the object is seen as a factor, but R deals with it as an integer. It is then possible to extract the list of levels with `levels(object)`.

```
d <- factor("True")
levels(d)
```

```
[1] "True"
```

R is able to convert some types of objects to others 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
```

If it is not possible for R to convert the object to the desired type during an operation, you will get an error.

1.1.3 Assessing the type of an object

In some logical operation, 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. This function will return a logical value (`TRUE` or `FALSE`) depending on the type of the object.

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

```r
b <- factor("R")
is.integer(b)
```

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

1.1.4 Changing the type of an object

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

- `as.integer()` will take data that *looks like integers* but are formatted as another type and change it to `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.

1.1.5 Mathematical operations on R objects

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

```r
# Addition and Subtraction
2 + 2

[1] 4

# Multiplication and Division
2 * 2 + 2 / 2

[1] 5

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

[1] 4.693147
```

1.1.6 Logical operations on R objects

You can also evaluate logical expressions in R:

```r
# Less than
5 < 6

[1] TRUE

# Greater than or equals to
5 >= 6

[1] FALSE

# Equals
5 == 6

[1] FALSE
```

```
# Not equals
5 != 6
```

```
[1] TRUE
```

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

```
[1] TRUE
```

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

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

```
[1] TRUE
```

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

```
[1] FALSE
```

1.2 Data structures

In Stata, data is stored into one dataset, saved as a dta file. In R, you have many other possibilities. The standard structures in R (they can all coexist in the same environment) are: vectors, matrices, and dataframes. Other composite structures exist, such as tibbles, but we will focus for now on the four basic ones.

1.2.1 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 basically 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.
- 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)`.

1.2.1.1 Creating vectors

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

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

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

# Display `temps` vector
temps
```

```
[1] 13 18 17 20 21
```

- Because atomic vectors are of a single type, if you assign values of different types to a vector, it 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`?

1.2.1.2 Naming 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("1st element", "2nd element",
                  "3rd element", "4th element")

# Display `vec_a` vector
vec_a

1st element 2nd element 3rd element 4th element
        "a"         "b"         "c"         "d"
```

1.2.1.3 Extracting a specific element of a vector

If you would like to extract a specific element of a vector into another object, you simply 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

3rd element
        "c"
```

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

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

# display vec_b
vec_b

3rd element
        "c"
```

1.2.1.4 Subsetting vectors

Subsetting is about extracting specific elements of an object. There are multiple ways of subsetting data in R. One of the easiest methods for vectors is to put the subset condition in the brackets:

```
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 is under 18, so it is removed).

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

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

vec_subensemble <- c(2, 8, 32)

vec_c[vec_c %in% vec_subensemble]
```

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

1.2.1.5 Changing and assessing the type of a vector

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 element. If it is not possible 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)`?

1.2.1.5.1 The case of factors

- `as.factor()` will reformat a vector into a factor object. Each unique element of the vector will be used as a level of the factor, in alphabetical order.
- Because the alphabetical order may not be the best suitable for you, you may want to define the factor levels yourself, with the `factor()` function. This is useful for ordinal variables, such as Likert scales.

```
vec_e <- c("very low", "low", "low", "very high", "high", "medium")

test_1 <- as.factor(vec_e)
levels(test_1)


test_2 <- factor(vec_e,
                 levels  = c("very low", "low",
                             "medium",
                             "high", "very high"),
                 ordered = TRUE)
levels(test_2)
```

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

1.2.1.6 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 length 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?

1.2.2 Lists

A list is a vector that can store elements of different types and of different length. 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).

1.2.2.1 Creating 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(vec_f = vec_f,
               vec_i = vec_i,
               vec_j = vec_j)
```

```
# Display the list
list_a

$vec_f
[1]  8 10 10 15 16

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

$vec_j
[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.

1.2.2.2 Extracting elements of 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"]]

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

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[2]).
- If the value to extract does not exist, NA is returned.

1.2.3 Matrices

- Data in a 2-dimensional structure can be represented in two formats, as a `matrix` or as 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.

[2] 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.

- 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.

1.2.3.1 Creating a matrix from scratch

A simple example of creating a matrix would be:

```
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`

```
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 purposes. 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 elements of the matrix. Only here, instead of using `names()`, we use `rownames()` and `colnames()`.

```
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
```

1.2.3.2 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 $a_1 b_1$, $a_2 b_2$ 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.

```
# 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
```

1.2.4 Dataframes

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

- Each vector is stored as a column of the dataframe.
- Each column as 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 from matrices, manually.
- Note: some packages create an object called *tibble*. The tibble is an alternative to the `dataframe` that has been created as a part of the `tidyverse` package. As `dataframe`, `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 `dataframe` and a `tibble`, but some functions in R require the use of `tibble`. You can simply convert `dataframe` into `tibble` with the `as.tibble()` function.

1.2.4.1 Creating a data frame

- Creating a dataframe might be as simple as converting a matrix into a dataframe:

```
df <- as.data.frame(m)

df

        V1 V2 V3
1st row 18 21 31
2nd row 10  8  6
```

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

```
vec_m <- c("a", "b")

df <- data.frame(vec_m, m)

df

        vec_m X1 X2 X3
1st row     a 18 21 31
2nd row     b 10  8  6
```

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

1.2.4.2 Adding a new column to a data frame

Now, in the example above, when we merge `vec_m` with the matrix m to create our dataframe, we assumed that the observations in the vector and in the matrix were following the same order. Imagine that you want to append a vector to a dataframe that is following a different order (elements of the new vector appear in a different order as the elements in our dataframe). It is still possible to merge them if you have a key by which you can match the rows.

Here is an example. Let's first create the following dataframe:

```
#
df_2 <- data.frame(id = c("b", "a", "c"),
                   value = c(15, 32, 81))
```

Let's assume that `id`, in `df_2`, and `vec_m`, in `df`, contains the same information: a unique identifier of each observation. You will note that `df_2` contains one more observations (`id = "c"`), and the order of observations differs from the ones in `df`. To merge nonetheless `df` with `df_2`, you can use the following function:

```
df_3 <- merge(x = df, y = df_2,
              by.x = "vec_m",
              by.y = "id",
              all.x = TRUE, all.y = TRUE)
```

- In the `merge()` function, the x and y indicate the names of the dataframe to combine.

- by.x and by.y, respectively indicate the names of the column to use to match observations between x and y.
- all.x and all.y indicate what to do with observations that do not appear in both x and y (here, "c" only appears in df_2).
  - If both are set to TRUE, all observations will be kept in the output dataframe, and missing columns will be filled with NA.
  - If one is set to FALSE, only the observations that also appear in the other dataframe will be kept.

1.2.4.3 Adding rows to a data frame

Let's assume that we would like to append the following dataframe to df_3:

```
df_4 <- data.frame(vec_m = c("e", "f"),
                   X1 = c(81, 32),
                   X2 = c(156, 32),
                   X3 = c(321, 51),
                   value = c(4, 127))
```

You can append the two new rows with the following function[3]:

```
df_3 <- rbind(df_3, df_4)
```

Note that the order of the columns do not matter with rbind, as long as they are all present.

1.2.4.4 Selections in a dataframe

Once you have a dataframe, you will usually want to create or manipulate particular columns of it. The columns are usually variables.

- The default way of invoking a named column in R is by appending a dollar sign and the column name to the data object. df$vec_m would return the variable named vec_m in the dataframe.
- Another way is to name the column of interest into brackets, as for vectors, lists, or matrices df[, "vec_m"]
  - Note the comma , before the variable name. This is because the dataframe is two-dimensional: rows are accessed first, then columns (yes, this is the contrary as for list, and yes, it is unpleasant). As such, if you want to extract the second elements of vec_m, you would write: df[2, "vec_m"].
  - Another way to extract the second element of vec_m: df$vec_m[2].
- You can also select a subset of columns or rows that meet a given condition

[3] Note that when you use the rbind function, all columns must be present in both dataframe. If you would like to append rows from a dataframe with missing columns, you can use the command rbind.fill from the plyr package. We will see during the next module how to install and use packages.

1.2.4.5 Subsetting a data frame

```
# Subsetting observations when X3 >= 35 AND X2 > 20
df_5 <- df_3[(df_3$X3 >= 35) & (df_3$X2 > 20),]
df_5
```

```
   vec_m X1  X2  X3 value
NA  <NA> NA  NA  NA    NA
4      e 81 156 321     4
5      f 32  32  51   127
```

Note that the column argument is left empty, so all columns are returned by default. You could as well indicate a vector of columns to keep in the output.

```
# Selecting specific columns while subsetting
df_5 <- df_3[(df_3$X3 >= 35) & (df_3$X2 > 20), c("vec_m", "X2", "X3")]
df_5
```

```
   vec_m  X2  X3
NA  <NA>  NA  NA
4      e 156 321
5      f  32  51
```

Now, you note that a line with missing values was added in the operation because `df_3` contains a row with missing information. You can subset incomplete observations with the following command:

```
# Excluding lines with missing values
df_5 <- df_5[complete.cases(df_5), ]
```

1.3 Data and the tidyverse

1.3.1 Principles of tidy data

Rules for tidy data (from *R for Data Science*):

1. Each variable must have its own column.
2. Each observation must have its own row.
3. Each value must have its own cell.

1.4 Tidy datasets

Let's load again the data from the Module 1.

```
# Dataset in SAS format
pisa_sas <- rio::import("data/cy07_msu_sch_qqq.sas7bdat")
```

1.4.1 Extract variables from dataframe

Now, you can see the two data files loaded in dataframes your environment, with 21'903 observations. But the dataset is very big. If you are interested in specific variables, you can extract them into a new `dataframe`. For example, to extract, `"CNT"`, `"CNTSCHID"`, `"SC016Q01TA"`, `"SC155Q02HA"`, from `pisa_sas`:

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

1.4.2 Remove dataframe or variables

You created a `dataframe` called `df`, containing the variables you are interested in. You may want to remove the two prior datasets that you loaded, to save memory:

```
rm(pisa_spss, pisa_sas)
```

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:

```
df[, "CNTSCHID"] <- NULL
```

Question: Do you see any way to get the same results using the `$` operator?

1.4.3 Rename columns

```
# With base R
names(df)[names(df) == "SC155Q02HA"]<- "new_name"
```

1.4.4 Extract a random sample from a dataframe

`df` is very big. If you would like to work on a subset of data, you can extract a random sample from your dataset with the following code:

```
df <- df[sample(nrow(df), 5000), ]
```

where `5000` is the number of random rows that you want to extract.

1.4.5 Gathering data

If values for a single variable are spread across multiple columns (e.g. income for different years), gather 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 long. Example:

```
library(dplyr)


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", "Ga
ston",
                                         "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.

1.4.5.1 A parenthesis about pipes, denoted %>%

A famous function from the tidyverse is *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 to a second step. Example:

```
library(dplyr)

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

This is absolutely equivalent to writing:

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

But the code bellow 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.

1.4.6 Spreading data

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 wide. Example:

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

1.4.7 Group data

Creating summary statistics by group is another routine task. This is accommodated in the tidyverse using the `group_by()`. In base R, you can use `aggregate()`.

The arguments, in addition to the data set, are simply the grouping variables separated by commas.

For example, let's calculate the mean per gender:

```
# 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
```

About the `aggregate` function, 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.

1.4.8 Arrange (sort) data

If you want to sort your data by the values of a particular variable, you can do so as well with the `arrange()` function (tidyverse), or with the `sort()` function (base R).

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

1.4.9 Filter data

Filtering keeps observations (rows) based on conditions, just like using subset conditions in the row arguments of a bracketed subset. Example:

```
# Using brackets
wages[(wages$school > 10) & (wages$exper > 10),]
```

1.4.10 Mutate data

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

```
# With base R:
wages$expsq <- wages$exper^2
```

1.4.11 Recode variables

Along with renaming variables, recoding variables is another integral part of data wrangling. You can do this with or without the tidyverse. For example, to recode `sex` as a dummy variable

```
# With the tidyverse
wages$is_female <- wages$sex %>% recode("male" = 0,
                                         "female" = 1)

# With base R
wages$is_female <- as.integer(wages$sex == "female")
```

1.4.12 Identify missing values

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

```
a <- NA

is.na(a)
[1] TRUE
```

You can apply it to a vector:

```
missing_sc016Q01ta <- is.na(df$SC016Q01TA)
```

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(missing_sc016Q01ta)
```

```
missing_sc016Q01ta
FALSE  TRUE
18853  3050
```

1.4.12.1 Removing missing values

In the previous lesson, we removed the missing values using:

```
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 `SC016Q01TA`?

Solution:

```
df_2 <- df[!is.na(df$SC016Q01TA), ]
```

1.4.12.2 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 value that *should be* `NA` but are code using a different schema by using brackets:

```
# Replace 99-denoted missing data with NA

# With base R
wages[wages$school == 99, ] <- NA
```

You can check for (correctly-coded) missing-values using the `is.na()` function.

```
## Missing
table(is.na(wages$school))


FALSE
 3294
```

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.

1.5 Merging data

In the first module, we learnt 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 in the `data.frame`.

Let’s assume is it 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:

```
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 identifier 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. This is often the case to have unique identifiers in your data, such as respondent id, organization number, municipality, *etc.* For panel data, this will typically be both the personal/group identifier and a timing variable, for example Sweden in 2015 in a cross-country analysis.

Note that 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.

1.5.1 Using `merge()`

1.5.1.1 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`

1.5.1.2 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`.

1.5.1.3 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)
```

1.5.2 Extract missing rows

Sometimes, you may want to keep only observations that do NOT appear in both dataset. 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`.

1.5.3 Appending data

Finally, instead of joining different datasets for the same individuals, sometimes you want to join together files that are for different individuals within the same dataset. When join data where the variables for each dataset are the same, but the observations are different, this is called *appending* data.

If you want to append `df2_not_df1` to `df1_not_df2`, you can use `plyr::rbind.fill()`. Make sure that the column in common have the same names. The missing columns will be filled with `NA`:

```
# Make sure the same column have the same name
names(df2_not_df1)[names(df2_not_df1) == "identifier"] <- "id"

df <- plyr::rbind.fill(df1_not_df2, df2_not_df1)
```