

Introduction

Javier García-Bernardo (based on material by Gerko Vink)

Introduction to R and RStudio

Recap

This morning we have learned the basics of programming in R:

- How to use RStudio, R-scripts and R-notebooks
- How to assign numbers to variables (`<-`)
- Data types (elements)
 - character, numeric, integer, logical, factor
- Data structures: composed of data types
 - vector, matrix, list, data.frame
- Code is applied on those data types/structures to transform them

Quick detour: Naming conventions

- You can't have spaces in your variable names.

```
student_cleaned <- 3
```

- Some programming languages tell you how to handle this, but not R.
- I use `students_cleaned`. This is the standard of Python.
- Use any, but be consistent

```
student_cleaned <- 3 #studentCleaned
```

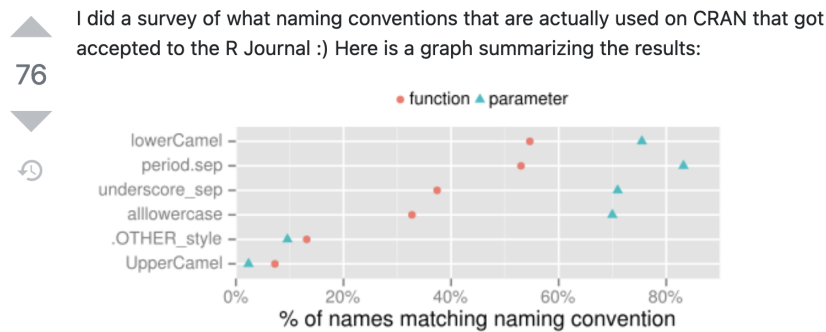


Figure 1: http://journal.r-project.org/archive/2012-2/RJournal_2012-2_Baaaath.pdf

Data Types (Elements)

Data Types

They are the basis of everything

1: Character

```
var_char <- "ab"
class(var_char)

## [1] "character"
```

2: Numeric

```
var_num <- 3.2
class(var_num)
```

```
## [1] "numeric"
```

```
var_ind <- "3"
class(var_ind)
```

```
## [1] "character"
```

```
converted_var <- as.numeric(var_ind)
class(converted_var)
```

```
## [1] "numeric"
```

```
var_inf <- Inf
class(var_inf)
```

```
## [1] "numeric"
```

```
var_nan <- NaN
class(var_nan)
```

```
## [1] "numeric"
```

3: Logical

```
var_true <- TRUE
class(var_true)
## [1] "logical"
```

Behind the scenes they are numbers: FALSE = 0, TRUE = 1

```
as.numeric(var_true)
## [1] 1
var_true + var_true
## [1] 2
```

They arise when comparing elements

```
a <- 3
b <- 5
a == b
## [1] FALSE

a == "3"
```

```
## [1] TRUE
```

```
a == 3
```

```
## [1] TRUE
```

Data Structures: Objects that contain more than one element

Basic data types (elements)

- character: “some text”
- numeric: e.g., 2.1
- integer: e.g., 2L
- logical: TRUE/FALSE
- factor: e.g., factor(“amsterdam”)

Basic data structures

- Consist of data types and functions to transform them
 - vector: `c(2, 4, 2)`
 - list: `list(first_col = 1, second = “a”, third = TRUE)`
 - matrix: `matrix(c(4, 4, 4, 4), nrow = 2, ncol = 2)`
 - data.frame: The most important ~ spreadsheet

1: Vector

- Collection of elements of the **same type**
- We concatenate the elements using the `c()` function (stands for concatenation)

```
#To create a vector we used `c()`, which stands for 'concatenation'.
```

```
a <- c(1, 2, 3, 4, 5)
```

```
a <- 1:5
```

```
a
```

```
## [1] 1 2 3 4 5
```

Characters (or character strings) in R are indicated by the double quote identifier.

```
c(a, "A")
```

```
## [1] "1" "2" "3" "4" "5" "A"
```

Repeating the same element many times

```
rep(a, 3)
```

```
## [1] 1 2 3 4 5 1 2 3 4 5 1 2 3 4 5
```

Retrieving elements from vectors by position

If we would want just the third element, we would type

```
#The first element in R is 1 (in other languages (e.g. Python) it is 0)
```

```
a[3]
```

```
## [1] 3
```

Retrieving elements from vectors by range

If we would want the first to the third (both included)

```
#The first element in R is 1 (in other languages (e.g. Python) it is 0)
a[1:3]
```

```
## [1] 1 2 3
```

Empty means all elements

```
#The first element in R is 1 (in other languages (e.g. Python) it is 0)
a[]
```

```
## [1] 1 2 3 4 5
```

2. Matrix: Multiple vectors in one object

This we would refer to as a matrix

```
matrix(3, nrow = 5, ncol = 2)
```

```
##      [,1] [,2]
## [1,]    3    3
## [2,]    3    3
## [3,]    3    3
## [4,]    3    3
## [5,]    3    3
```

#We can also create them with vectors, it fills the matrix by column. But be careful!

```
c <- matrix(c(1,2,3), nrow = 5, ncol = 2)
```

```
## Warning in matrix(c(1, 2, 3), nrow = 5, ncol = 2): data length [3] is not a sub-
## multiple or multiple of the number of rows [5]
```

```
c
##      [,1] [,2]
## [1,]    1    3
## [2,]    2    1
## [3,]    3    2
## [4,]    1    3
## [5,]    2    1
```

Retrieving elements in matrices

The first row is retrieved with

```
c[1, ]
## [1] 1 3
```

The second column is retrieved with

```
c[, 2]
## [1] 3 1 2 3 1
```

The intersection of the first row and second column is called by

```
c[1, 2]
## [1] 3
```

We can also use ranges.

```
c[1:3, 2]
## [1] 3 1 2
```

Matrices with mixed numeric / character data

If we add a character column to matrix `c`; everything becomes a character:

```
cbind(c, c("a", "b", "c", "d", "e"))
```

```
##      [,1] [,2] [,3]
## [1,] "1"  "3"  "a"
## [2,] "2"  "1"  "b"
## [3,] "3"  "2"  "c"
## [4,] "1"  "3"  "d"
## [5,] "2"  "1"  "e"
```

Remember, matrices and vectors are numerical OR character objects. They can never contain both and still be used for numerical calculations.

3. Data frames

Data frames can contain both numerical and character elements at the same time, although never in the same column. You can think of them as spreadsheets (with column names, and rows)

```
d <- data.frame("V1" = rnorm(5),
                 "V2" = rnorm(5, mean = 5, sd = 2),
                 "V3" = c("a", "a", "b", "b", "b"))
```

d

```
##      V1      V2 V3
## 1 0.5235026 9.063058 a
## 2 0.7361073 3.204876 a
## 3 -0.4284985 4.499260 b
## 4 -0.2207737 4.614878 b
## 5 0.1073378 1.155706 b
```

We ‘filled’ a dataframe with two randomly generated sets from the normal distribution - where $V1$ is standard normal and $V2 \sim N(5, 2)$ - and a character set.

Data frames (continued)

You can name the columns and rows in data frames (just like in matrices)

#Rows

```
row.names(d) <- c("row 1", "row 2", "row 3", "row 4", "row 5")
```

d

```
##      V1      V2 V3
## row 1 0.5235026 9.063058 a
## row 2 0.7361073 3.204876 a
## row 3 -0.4284985 4.499260 b
## row 4 -0.2207737 4.614878 b
## row 5 0.1073378 1.155706 b
```

#Columns

```
names(d) <- c("V1", "V2_new", "V3")
```

d

```
##      V1    V2_new V3
## row 1 0.5235026 9.063058 a
## row 2 0.7361073 3.204876 a
## row 3 -0.4284985 4.499260 b
```

```
## row 4 -0.2207737 4.614878 b
## row 5 0.1073378 1.155706 b
```

Retrieving row elements in data frames

There are two ways to obtain row 3 from data frame d:

```
d["row 3", ]
```

```
##           V1  V2_new V3
## row 3 -0.4284985 4.49926 b
```

and

```
d[3, ]
```

```
##           V1  V2_new V3
## row 3 -0.4284985 4.49926 b
```

CAREFUL! You always need the comma (,) when filtering by rows

Retrieving columns elements in data frames

Both

```
d[, "V2_new"] # and
```

```
## [1] 9.063058 3.204876 4.499260 4.614878 1.155706
```

```
d[, 2]
```

```
## [1] 9.063058 3.204876 4.499260 4.614878 1.155706
```

```
d$V2_new
```

```
## [1] 9.063058 3.204876 4.499260 4.614878 1.155706
```

Retrieving both

The intersection between row 2 and column 3 can be obtained by

```
d[2, 3]
```

```
## [1] "a"
```

And you can also use ranges

```
d[3:4, 1:2]
```

```
##           V1  V2_new
## row 3 -0.4284985 4.499260
## row 4 -0.2207737 4.614878
```

4. Factor

- A data type used to encode categorical variables
- It'll be useful for data modeling and data isualization

```
d$V3_factor = factor(d$V3, labels = c("Amsterdam", "Utrecht"))
d
##           V1  V2_new V3 V3_factor
```

```
## row 1  0.5235026 9.063058 a Amsterdam
## row 2  0.7361073 3.204876 a Amsterdam
## row 3 -0.4284985 4.499260 b  Utrecht
## row 4 -0.2207737 4.614878 b  Utrecht
## row 5  0.1073378 1.155706 b  Utrecht
```

Class

```
class(d$V3_factor)
## [1] "factor"
```

Summary

```
str(d$V3_factor)
## Factor w/ 2 levels "Amsterdam","Utrecht": 1 1 2 2 2
```

Use of as.xxx functions

You can convert between data types (as long as the conversion is valid)

```
d$V2_int = as.integer(d$V2_new)
d
```

```
##           V1    V2_new V3 V3_factor V2_int
## row 1  0.5235026 9.063058 a Amsterdam     9
## row 2  0.7361073 3.204876 a Amsterdam     3
## row 3 -0.4284985 4.499260 b  Utrecht     4
## row 4 -0.2207737 4.614878 b  Utrecht     4
## row 5  0.1073378 1.155706 b  Utrecht     1
```

```
d$V2_char = as.character(d$V2_int)
d
```

```
##           V1    V2_new V3 V3_factor V2_int V2_char
## row 1  0.5235026 9.063058 a Amsterdam     9        9
## row 2  0.7361073 3.204876 a Amsterdam     3        3
## row 3 -0.4284985 4.499260 b  Utrecht     4        4
## row 4 -0.2207737 4.614878 b  Utrecht     4        4
## row 5  0.1073378 1.155706 b  Utrecht     1        1
```

5. List: Mixed type

You can have a list of everything mixed with everything. For example, an simple list can be created by

```
a <- 1:5 #a vector
f <- list(a) #convert to a list
f
```

```
## [[1]]
## [1] 1 2 3 4 5
```

Elements or objects within lists can be called by using double square brackets `[[]]`. For example, the first (and only) element in list `f` is object `a`

```
f[[1]]
```

```
## [1] 1 2 3 4 5
```

Lists (continued)

We can simply add an object or element to an existing list

```
f[[2]] <- d
f

## [[1]]
## [1] 1 2 3 4 5
##
## [[2]]
##           V1    V2_new V3 V3_factor V2_int V2_char
## row 1  0.5235026 9.063058 a Amsterdam     9       9
## row 2  0.7361073 3.204876 a Amsterdam     3       3
## row 3 -0.4284985 4.499260 b  Utrecht     4       4
## row 4 -0.2207737 4.614878 b  Utrecht     4       4
## row 5  0.1073378 1.155706 b  Utrecht     1       1
```

to obtain a list with a vector and a data frame.

Lists (continued)

We can add names to the list as follows

```
names(f) <- c("vector", "data frame")
f

## $vector
## [1] 1 2 3 4 5
##
## $`data frame`
##           V1    V2_new V3 V3_factor V2_int V2_char
## row 1  0.5235026 9.063058 a Amsterdam     9       9
## row 2  0.7361073 3.204876 a Amsterdam     3       3
## row 3 -0.4284985 4.499260 b  Utrecht     4       4
## row 4 -0.2207737 4.614878 b  Utrecht     4       4
## row 5  0.1073378 1.155706 b  Utrecht     1       1
```

Lists (continued)

We can also create it as

```
f <- list("vector" = 1:5,
          "data frame" = f)
f

## $vector
## [1] 1 2 3 4 5
##
## $`data frame`
## $`data frame`$vector
## [1] 1 2 3 4 5
##
## $`data frame`$`data frame`
##           V1    V2_new V3 V3_factor V2_int V2_char
## row 1  0.5235026 9.063058 a Amsterdam     9       9
## row 2  0.7361073 3.204876 a Amsterdam     3       3
## row 3 -0.4284985 4.499260 b  Utrecht     4       4
```



```
## row 4 -0.2207737 4.614878 b Utrecht 4 4
## row 5 0.1073378 1.155706 b Utrecht 1 1
```

Retrieving elements in lists

Retrieving the vector from the list can be done as follows

```
f[[1]]
```

```
## [1] 1 2 3 4 5
```

```
f[["vector"]]
```

```
## [1] 1 2 3 4 5
```

```
f$vector
```

```
## [1] 1 2 3 4 5
```

Lists in lists

```
g <- list(1:4, list("vector" = 5:1,
                    "matrix_ex" = matrix(0, nrow = 2, ncol = 2)))
g
```

```
## [[1]]
```

```
## [1] 1 2 3 4
```

```
##
```

```
## [[2]]
```

```
## [[2]]$vector
```

```
## [1] 5 4 3 2 1
```

```
##
```

```
## [[2]]$matrix_ex
```

```
##      [,1] [,2]
```

```
## [1,]    0    0
```

```
## [2,]    0    0
```

To call the vector from the second list within the list g, use the following code

```
g[[2]][[1]] #Also g[[2]]$vector
```

```
## [1] 5 4 3 2 1
```

Filtering by condition

Logical operators

- Logical operators are signs that evaluate a statement, such as == (equal), != (different), < (lower), > (greater), <= (lower or equal), >= (greater or equal), and | (OR) as well as & (AND).
- For example, if we would like elements out of example vector below that are larger than 3, we would type:

```
example_vector = c(1,2,3,4,5,6,7,8,9)
example_vector[example_vector>3]
## [1] 4 5 6 7 8 9
```

Why does this work? It filters the vector, keeping the elements where the condition is TRUE

```
example_vector > 3
## [1] FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
```

Logical operators (cont'd)

If we would like the elements that are smaller than 3 OR larger than 3, we could type.

```
example_vector[(example_vector < 3) | (example_vector > 3)] #c smaller than 3 or larger than 3

## [1] 1 2 4 5 6 7 8 9
```

or

```
example_vector[example_vector != 3] #c not equal to 3

## [1] 1 2 4 5 6 7 8 9
```

Logical operators (cont'd)

You can use %in to select specific elements

```
example_vector %in% c(1,5)

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

example_vector[example_vector %in% c(1,5)]

## [1] 1 5
```

Typing ! before a logical operator takes the complement of that action (the opposite)

```
!(example_vector %in% c(1,5))

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

Logical operators (cont'd)

They are extremely useful in dataframes

```
d
##           V1    V2_new V3 V3_factor V2_int V2_char
## row 1  0.5235026 9.063058 a Amsterdam     9      9
## row 2  0.7361073 3.204876 a Amsterdam     3      3
## row 3 -0.4284985 4.499260 b  Utrecht     4      4
## row 4 -0.2207737 4.614878 b  Utrecht     4      4
## row 5  0.1073378 1.155706 b  Utrecht     1      1
```

```
d$V1 > 0

## [1]  TRUE  TRUE FALSE FALSE  TRUE

d[d$V1 > 0, ]
```

```
##           V1    V2_new V3 V3_factor V2_int V2_char
## row 1  0.5235026 9.063058 a Amsterdam     9      9
## row 2  0.7361073 3.204876 a Amsterdam     3      3
## row 5  0.1073378 1.155706 b  Utrecht     1      1
```

Functions

Basic idea

- Take some standard input (e.g. a vector of numbers)
- Return some standard output (e.g. the mean)
- You call them with parenthesis

```
mean(c(1,2,3,4,5))  
## [1] 3
```

- You can save the output with a name

```
mean_v <- mean(c(1,2,3,4,5))  
mean_v  
## [1] 3
```

- When they come from a package, you can call them in two ways

```
#install.packages("dplyr")  
library(dplyr)  
#You can specify the package where the library comes from  
dplyr::n_distinct(c(1,2,2,3,3))
```

Function's arguments

- Functions typically have arguments (i.e., things that the function need to work).
- Usually at least one is required. e.g., in the mean example you need to pass a vector of numbers.
- Other arguments tell the function how it should handle the input. In the example below it tells the mean function to remove missing values.

```
mean(c(1,2,3,4,5,NA))  
## [1] NA  
mean(c(1,2,3,4,5,NA),na.rm = TRUE)  
## [1] 3
```

- You can create your own (more on this next day)

```
my_mean <- function(vector_num){  
  return(sum(vector_num)/length(vector_num))  
}  
  
my_mean(c(1,2,3,4,5))  
## [1] 3
```

Math errors

Things that cannot be done

- Things that have no representation in real number space (at least not without tremendous effort)
 - For example, the following code returns “Not a Number”

```
0 / 0  
## [1] NaN
```

- Also impossible are calculations based on missing values (NA's)

```
mean(c(1, 2, NA, 4, 5))  
## [1] NA
```

- You can ignore missing values (often not recommended). Two ways:

```
mean(c(1, 2, NA, 4, 5), na.rm = TRUE)
## [1] 3
mean(na.omit(c(1, 2, NA, 4, 5)))
## [1] 3
```

Be careful with rounding errors

- The computer uses approximations of the numbers (floating-point arithmetic). This can create problems in R:

```
(3 - 2.9)
## [1] 0.1
(3 - 2.9) == 0.1
## [1] FALSE
(3 - 2.9) - .1
## [1] 8.326673e-17
```

You can use the function `near` from the library `dplyr`

```
#install.packages("dplyr")
#library(dplyr) #we already run this earlier, we don't need it again
dplyr::near((3 - 2.9), 0.1)

## [1] TRUE
```

Applications

Reading a CSV file and calculating descriptive statistics

That was dry, but let's see some potential, reading a CSV file and calculating some descriptive statistics.

You'll need to install the packages/load the library beforehand

```
#install.packages(c("readr", "readxl", "haven", "foreign", "Hmisc"),
#repos = "http://cran.us.r-project.org")

readr::read_delim("here_path.csv", delim = ",") #CSV
readr::read_rds("here_path.Rds") #RDS (R format)
readxl::read_excel("here_path.xlsx") #Excel
haven::read_dta("here_path.dta") #Stata
haven::read_spss("here_path.sav") #SPSS

#Read example data
df <- readr::read_delim("../common_datasets/dataset_boys.csv", delim = ",")
## Rows: 748 Columns: 9
## -- Column specification -----
## Delimiter: ","
## chr (3): gen, phb, reg
## dbl (6): age, hgt, wgt, bmi, hc, tv
##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

Calculate some descriptive statistics

#Give some information

```
str(df)
```

```
## spec_tbl_df [748 x 9] (S3: spec_tbl_df/tbl_df/tbl/data.frame)
## $ age: num [1:748] 0.035 0.038 0.057 0.06 0.062 0.068 0.068 0.071 0.071 0.073 ...
## $ hgt: num [1:748] 50.1 53.5 50 54.5 57.5 55.5 52.5 53 55.1 54.5 ...
## $ wgt: num [1:748] 3.65 3.37 3.14 4.27 5.03 ...
## $ bmi: num [1:748] 14.5 11.8 12.6 14.4 15.2 ...
## $ hc : num [1:748] 33.7 35 35.2 36.7 37.3 37 34.9 35.8 36.8 38 ...
## $ gen: chr [1:748] NA NA NA NA ...
## $ phb: chr [1:748] NA NA NA NA ...
## $ tv : num [1:748] NA NA NA NA NA NA NA NA NA NA ...
## $ reg: chr [1:748] "south" "south" "south" "south" ...
## - attr(*, "spec")=
## .. cols(
## ..   age = col_double(),
## ..   hgt = col_double(),
## ..   wgt = col_double(),
## ..   bmi = col_double(),
## ..   hc = col_double(),
## ..   gen = col_character(),
## ..   phb = col_character(),
## ..   tv = col_double(),
## ..   reg = col_character()
## .. )
## - attr(*, "problems")=<externalptr>
```

```
summary(df)
```

```
##      age      hgt      wgt      bmi
## Min.   : 0.035   Min.   : 50.00   Min.   :  3.14   Min.   :11.77
## 1st Qu.: 1.581   1st Qu.: 84.88   1st Qu.: 11.70   1st Qu.:15.90
## Median :10.505   Median :147.30   Median : 34.65   Median :17.45
## Mean   :  9.159   Mean   :132.15   Mean   : 37.15   Mean   :18.07
## 3rd Qu.:15.267   3rd Qu.:175.22   3rd Qu.: 59.58   3rd Qu.:19.53
## Max.   :21.177   Max.   :198.00   Max.   :117.40   Max.   :31.74
##      NA's      :20      NA's      :4      NA's      :21
##      hc      gen      phb      tv
## Min.   :33.70   Length:748   Length:748   Min.   :  1.00
## 1st Qu.:48.12   Class :character   Class :character   1st Qu.:  4.00
## Median :53.00   Mode  :character   Mode  :character   Median :12.00
## Mean   :51.51
## 3rd Qu.:56.00
## Max.   :65.00
## NA's   :46
##      reg
## Length:748
## Class :character
## Mode  :character
##
##
```

```
##  
##
```

```
Hmisc::describe(df)
```

```
## df  
##  
## 9 Variables      748 Observations  
## -----  
## age  
##      n missing distinct      Info      Mean      Gmd      .05      .10  
##    748      0      683        1    9.159    7.827    0.151    0.271  
##    .25    .50    .75    .90    .95  
##   1.581   10.505   15.267   17.982   19.091  
##  
## lowest : 0.035 0.038 0.057 0.060 0.062, highest: 20.429 20.761 20.780 20.813 21.177  
## -----  
## hgt  
##      n missing distinct      Info      Mean      Gmd      .05      .10  
##    728      20      482        1   132.2    52.59    58.00    62.85  
##    .25    .50    .75    .90    .95  
##   84.88   147.30   175.22   183.90   188.00  
##  
## lowest : 50.0 50.1 52.5 53.0 53.5, highest: 195.5 195.7 196.2 196.7 198.0  
## -----  
## wgt  
##      n missing distinct      Info      Mean      Gmd      .05      .10  
##    744      4      523        1    37.15    29.55    5.082    6.692  
##    .25    .50    .75    .90    .95  
##   11.700   34.650   59.575   71.970   79.425  
##  
## lowest : 3.14 3.37 3.65 3.73 3.81, highest: 99.00 100.10 102.00 113.00 117.40  
## -----  
## bmi  
##      n missing distinct      Info      Mean      Gmd      .05      .10  
##    727      21      503        1    18.07    3.291    14.28    14.83  
##    .25    .50    .75    .90    .95  
##   15.90   17.45   19.53   22.44   23.86  
##  
## lowest : 11.77 12.33 12.56 12.77 12.78, highest: 29.03 29.93 30.62 31.34 31.74  
## -----  
## hc  
##      n missing distinct      Info      Mean      Gmd      .05      .10  
##    702      46      204        1    51.51    6.504    39.00    41.82  
##    .25    .50    .75    .90    .95  
##   48.12   53.00   56.00   57.80   58.70  
##  
## lowest : 33.7 34.9 35.0 35.2 35.7, highest: 60.0 60.2 60.3 60.5 65.0  
## -----  
## gen  
##      n missing distinct  
##    245      503      5  
##
```

```

## lowest : G1 G2 G3 G4 G5, highest: G1 G2 G3 G4 G5
##
## Value      G1      G2      G3      G4      G5
## Frequency   56     50     22     42     75
## Proportion 0.229 0.204 0.090 0.171 0.306
## -----
## phb
##      n missing distinct
##    245     503         6
##
## lowest : P1 P2 P3 P4 P5, highest: P2 P3 P4 P5 P6
##
## Value      P1      P2      P3      P4      P5      P6
## Frequency   63     40     19     32     50     41
## Proportion 0.257 0.163 0.078 0.131 0.204 0.167
## -----
## tv
##      n missing distinct      Info      Mean      Gmd      .05      .10
##    226     522         18    0.988    11.89    9.116         2         2
##      .25      .50      .75      .90      .95
##        4        12        20        25        25
##
## lowest :  1  2  3  4  5, highest: 16 17 18 20 25
##
## Value      1      2      3      4      5      6      8      9     10     12     13
## Frequency    5     26     19     17      5     10     13      1     16     15      1
## Proportion 0.022 0.115 0.084 0.075 0.022 0.044 0.058 0.004 0.071 0.066 0.004
##
## Value      14     15     16     17     18     20     25
## Frequency    1     27      1      1      1     38     29
## Proportion 0.004 0.119 0.004 0.004 0.004 0.168 0.128
## -----
## reg
##      n missing distinct
##    745         3         5
##
## lowest : city east north south west , highest: city east north south west
##
## Value      city east north south west
## Frequency   73   161     81   191   239
## Proportion 0.098 0.216 0.109 0.256 0.321
## -----

```

Some programming tips:

- **Keep your code clean**
 - Break the code in components, keep it tidy
 - Use (at least) a folder for the data, and another for figures; don't save all code in one folder.
 - If you have several R files, use descriptive names (e.g. 1_data_collection.Rmd; 2_data_cleaning.Rmd; etc)
 - Write all code in the source editor, don't use the console until you know what you are doing. Otherwise you'll forget to copy a step to the code and you'll not be able to remember what you did.
- Use **comments** (text preceded by #) to clarify what you are doing

- If you look at your code again, one year from now: you will not know what you did -> unless you use comments

Useful shortcuts

- **Tab** while typing in the console: list all objects with that name
- **Ctrl+Enter** (Windows) or **Cmd+Enter** (Mac): run line or selection
- **Ctrl+Alt+I** (Windows) or **Cmd+Option+I** (Mac): insert R chunk

Practical

How to approach the next practical

Aim to make the exercises without looking at the answers.

- Use the answers to evaluate your work
- Use the help to identify how functions work

If this does not work out -> show the code.

In any case; ask for help when you feel help is needed.

- Do not ‘struggle’ for too long: we only have limited time!