Chapter 1 - A Primer

Dr. Alessandro Ruggieri

Contents

Fundamentals

R is a free statistical software focused on manipulating and analyzing data. Everything in R is an object: e.g. datasets, functions, models, plots, etc. R is open source and it is highly extendable, i.e. anyone can contribute to the R project,develop and distribute code to run on the R platform.

Download and Install

To download and install R, go to the CRAN homepage (https://cran.r-project.org/) and following links to download and install.

To download and install RStudio, go to the RStudio webpage: https://rstudio.com/products/rstudio/

Set Directory Path

If you want to change the directory you can run setwd( ) and include the path you want. For example:

```
setwd("/Users/alessandroruggieri/Documents/data_lab")
```

To see what the current working directory is set to, run the command getwd( ).

```
getwd()
```

```
## [1] "/Users/alessandroruggieri/Documents/data_lab"
```

Comments

Text that are preceded by the symbol # will be treated as a comment by R, i.e. they will be executed it as code.

```
# This is a comment line and it won't be executed by R
```

Data structures

R operates using data structures, which include vectors (a sequence of numerical, character or logical data), matrices (multidimentional collection of vectors of the same type) and data frame (multidimentional collection of possibly different data types)

R as a Calculator

R has a lot of basic math functionality built into it. For example, it has all of the ordinary arithmetic operation. Notationwise, infinity is Inf, and negative infinity is -Inf. NaN governs not a number, which is to be distinguished from NA (a missing value), or NULL, the null object.

Assigning Objects

To store a structure in the computer's memory, we have to assign it to an object. Objects can be assigned by using the operator "<-". For instance, we can write

```
# integer
y <- 1
#  double
y <- 1.3
# string
x <- "data lab"
# boolean
x <- TRUE
```

Remember, do not use equals (=) for assignment.

Vectors

A vector in R is sequence of elements of the same type. The concatenate function, c( ), can be used to join data from end to end to create vectors. Vectors can be numeric

```
# create a numerical vector called "v"
v <-c(1,4,15,6,10)
```

or characters/string

```
# create a string vector called "names"
names <-c("John","Emily")
```

Once a vector is saved as an object (i.e. variable), you can access different parts of the vector by referencing its indexed position using squared brackets:

```
# access second entry of string vector "names"
names[2]
```

```
## [1] "Emily"
```

You can also explore the structure of a vector using the structure function, str( ):

```
# display structure of string vector "names"
str(names)
```

```
##  chr [1:2] "John" "Emily"
```

The sequence function, seq( ), can be used to create an equidistant series of values. For instance, we can generate a sequence of numbers from 1 to 10 in increments of 1 (vector a), or a sequence of numbers from 1 to 10 of length 15 (vector b):

```
# create a numerical sequence from 1 to 10 with step size=2
a<-seq(1,10, by=1)
# create a numerical sequence from 1 to 10 containing n=15 equidistant points
b<-seq(1,10, len=15)
```

Factors

Categorical variables in R are stored as "factors". The factor function takes vectors of any type and convert them to factors. For instance:

```
# create a vector called 'gender'
gender <- c("f", "f", "f", "m", "m", "m", "m")
# transform 'gender' into a factor object
gender <- factor(gender)
# examine the structure of 'gender'
str(gender)
```

```
##  Factor w/ 2 levels "f","m": 1 1 1 2 2 2 2
```

List

A list is a sequence of elements of different types. For instance, we can combine three vectors, each of a different type, into a single list:

```
# create a list of three different vectors
l <- list(x=x, names=names, gender=gender)
```

You can call specific elements within the list using the list index within squared brackets as follows:

```
# call the second element from the list
l[[2]]
```

```
## [1] "John"  "Emily"
```

or we can examine the structure of a specific elements of the list as follows:

```
# examine the structure of second element in the list
str(l[[2]])
```

```
##  chr [1:2] "John" "Emily"
```

Data frames and matrices

Data frames are two dimensional objects. You can manually create data frames by combining two vectors using the function data.frame( )

```
# vectors of exam names
modules <- c("Math 1","Math 2","Stats 1",
             "Stats 2","Macro 1","Macro 2",
             "Micro 1", "Micro 2","Labor",
             "Trade","Monetary","Behavioral",
             "Econometrics", "PolEcon", "Finance",
             "Development", "Experimental")
# vectors of exam marks
marks <- c(7.4, 8.2, 6.5, 7, 6, 7, 5.8, 6.4, 4.2, 5.3, 6.4, 6.5, 4.9,5.9, 7.2, 9.1, 5.8)
# mark sheet
marksheet <- data.frame(modules, marks)
```

The names of the columns (or variables) are stored as attributes of the data frame and can be called using the function names( ):

```
# names of the variables stored in the mark sheet
names(marksheet)
```

```
## [1] "modules" "marks"
```

The columns of a data frame can be renamed using the function names( ) as follows:

```
# rename column vectors of mark-sheet
names(marksheet) <- c("Modules", "Marks")
```

The columns of a data frame can also be name when we first crate the data frame by assigning the name to each vector of data as follows:

```
# create mark sheet
marksheet <- data.frame("Modules"=modules, "marks"=marks)
```

The vectors of a data frame can be accessed either using "$" and specifying the name of the vector we want to display, i.e.

```
# Access the vector "Modules" of our data frame
marksheet$Modules
```

```
##  [1] "Math 1"       "Math 2"       "Stats 1"      "Stats 2"      "Macro 1"
##  [6] "Macro 2"      "Micro 1"      "Micro 2"      "Labor"        "Trade"
## [11] "Monetary"     "Behavioral"   "Econometrics" "PolEcon"      "Finance"
## [16] "Development"  "Experimental"
```

or by specifying the selected column into squared brackets

```
# Access the first vector of our data frame
marksheet[,1]
```

```
##  [1] "Math 1"       "Math 2"       "Stats 1"      "Stats 2"      "Macro 1"
##  [6] "Macro 2"      "Micro 1"      "Micro 2"      "Labor"        "Trade"
## [11] "Monetary"     "Behavioral"   "Econometrics" "PolEcon"      "Finance"
## [16] "Development"  "Experimental"
```

Similarly, we can access a row in a data frame, by specifying it into squared brackets

```
# Access the first row of our data frame
marksheet[1,]
```

```
##   Modules marks
## 1  Math 1   7.4
```

Notice that matrix is similar to a data frame except that all of its values are numeric and there are no strings.

Flow Control

R supports standard flow control structures used in programming:

- If-Then-Else
- For Loop
- While Loop

If-Then-Else

If-Then-Else is programming language statement that compares two or more sets of data and tests the results. f the results are true, the THEN instructions are taken; if not, the ELSE instructions are taken

```
if (marks[1] > 5) {
    print("Marks in Exam 1 larger than 5")
} else {
    print("Marks in Exam 1 lower than 5")
}
```

```
## [1] "Marks in Exam 1 larger than 5"
```

For Loop

A For Loop enables a particular set of conditions to be executed repeatedly until a condition is satisfied.

```
# Print names of modules using for loop
for (value in modules) {
    print(value)
}
```

```
## [1] "Math 1"
## [1] "Math 2"
## [1] "Stats 1"
## [1] "Stats 2"
## [1] "Macro 1"
## [1] "Macro 2"
## [1] "Micro 1"
```

```
## [1] "Micro 2"
## [1] "Labor"
## [1] "Trade"
## [1] "Monetary"
## [1] "Behavioral"
## [1] "Econometrics"
## [1] "PolEcon"
## [1] "Finance"
## [1] "Development"
## [1] "Experimental"
```

While Loop

A while loop is a control flow statement that allows code to be executed repeatedly based on a given Boolean condition. The while loop can be thought of as a repeating if statement.

```
# while loop
i<-0
while (i < length(marks)) {
  print(modules[i])
  print(marks[i])
  i<-i+1
}
```

```
## character(0)
## numeric(0)
## [1] "Math 1"
## [1] 7.4
## [1] "Math 2"
## [1] 8.2
## [1] "Stats 1"
## [1] 6.5
## [1] "Stats 2"
## [1] 7
## [1] "Macro 1"
## [1] 6
## [1] "Macro 2"
## [1] 7
## [1] "Micro 1"
## [1] 5.8
## [1] "Micro 2"
## [1] 6.4
## [1] "Labor"
## [1] 4.2
## [1] "Trade"
## [1] 5.3
## [1] "Monetary"
## [1] 6.4
## [1] "Behavioral"
## [1] 6.5
## [1] "Econometrics"
## [1] 4.9
## [1] "PolEcon"
## [1] 5.9
```

```
## [1] "Finance"
## [1] 7.2
## [1] "Development"
## [1] 9.1
```

Functions

Functions are pieces of code performing a specific task using specific inputs. In R, a function consists of a function name and arguments used as inputs, and it returns a specific output.

Some R base functions

R has many built in functions and additional functions can be installed and loaded. Some helpful base functions in R are:

- the function length( ) counts the number of values in a vector

```
# Number of marks
nummark <- length(marks)
# Print result
sprintf("Number of marks stored: %f", nummark)
```

```
## [1] "Number of marks stored: 17.000000"
```

- the function sum( ) takes value from a vector and compute their sum

```
# Sum of the marks
totmark <- sum(marks)
# Print result
sprintf("Sum of marks: %f", totmark)
```

```
## [1] "Sum of marks: 109.600000"
```

Custom functions

We can construct our own custom function that may or may not take arguments or return a result.

For instance, we can write a function that to calculate how many values are in a vector as follows:

```
# Functions that computes the length values from a vector x
mylength <- function(x) {

  out<-0
  for (value in x) {
      out <- out++1
  }

  return(out)
}
# Compute average marks using own-built function
mylengthmark <-mylength(marks)
# Print result
sprintf("Lenght of vector of marks (computed using own-built function): %f", mylengthmark)
```

```
## [1] "Lenght of vector of marks (computed using own-built function): 17.000000"
```

This function takes a vector or set of numbers as its arguments, and count how many elements the vector is composed of iteratively. To check if our function produces the right output, we compare the outcomes as follows:

```
# Check if own-built function is correct
length(marks) == mylength(marks)
```

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

Another example is a function that calculates the sum of the values from an input vector as follows:

```
# Functions that computes the sum values from a vector x
mysum <- function(x) {

  n<-length(x)
  i<-1
  out<-0

  while (i < n+1) {
    out<-out+x[i]
    i<-i+1
  }

  return(out)
}
# Compute sum of marks using own-built function
mysummark <-mysum(marks)
# Print result
sprintf("Sum of marks (computed using own-built function): %f", mysummark)
```

```
## [1] "Sum of marks (computed using own-built function): 109.600000"
```

This function takes a vector or set of numbers as its arguments, and sums those numbers adding their values iteratively. Again, we check if the function gives the right output:

```
# Check if own-built function is correct
round(sum(marks),10) == round(mysum(marks),10)
```

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

Notice that we rounded the outcomes of the function to the 10th decimal. We do this to avoid discrepancies between the two functions due to how R stores scalars.

Help for functions

To get help for R, you can use the information contained in the R Project webpage (https://www.r-project.org/). If you already know the command you want help for, you can directly type help(command) in the console. Alternatively, you can get help with a function in R by using the question mark operator (?) followed by the function name.

```
# Help with sum function
help(sum)
?sum
```

You can also get example usage of a function by calling the example() function and passing the name of the function as an argument

```
# Example with sum function
example(sum)
```

```
##
## sum> ## Pass a vector to sum, and it will add the elements together.
## sum> sum(1:5)
## [1] 15
##
## sum> ## Pass several numbers to sum, and it also adds the elements.
## sum> sum(1, 2, 3, 4, 5)
## [1] 15
##
## sum> ## In fact, you can pass vectors into several arguments, and everything gets added.
## sum> sum(1:2, 3:5)
## [1] 15
##
## sum> ## If there are missing values, the sum is unknown, i.e., also missing, ....
## sum> sum(1:5, NA)
## [1] NA
##
## sum> ## ... unless  we exclude missing values explicitly:
## sum> sum(1:5, NA, na.rm = TRUE)
## [1] 15
```

Basic Math

Pseudo random number

Most of the major probability distributions are built into R:

- rbinom( ) generates pseudo-random numbers from a binomial distribution

```
# Generate 100 random numbers from binomial (0-1) with parameters n = 10 and p = 2
v<-rbinom(1, n=10, p=0.3)
```

- rexp( ) generates pseudo-random numbers from an exponential distribution

```
# Generate 100 random numbers from exponential with rate 3
v<-rexp(100, r=3)
```

- rnorm( ) generates pseudo-random numbers from a normal distribution

```
# Generate 100 random numbers from a normal with mean=0 and sd=3
v<-rnorm(100, mean=0, sd=3)
```

Numerical integration

Numerical integration of a function is done with the integrate( ) function. Consider the function $y = \exp(-x)$

```
## PDF of exponential random variable
f <- function (x) exp(-x)
```

We can compute the integral of this funtion between 0 and 1 as follows:

```
# integrate PDF exponential random variable
prob<-integrate(f, lower = 0, upper = 1)
```

The output value of integrate, like so many other functions in R, is a named list

```r
# integration outcome structure
str(prob)
```

```
## List of 5
##  $ value       : num 0.632
##  $ abs.error   : num 7.02e-15
##  $ subdivisions: int 1
##  $ message     : chr "OK"
##  $ call        : language integrate(f = f, lower = 0, upper = 1)
##  - attr(*, "class")= chr "integrate"
```

Optimization

R contains also a number of routines for optimization. The main optimization function that comes with R is optim( ). By default, it performs minimization. optim( ) accepts two arguments: a function and a starting point for the optimization routine (in the opposite order). Note that the function must have a vector argument, like f(x) where x is a vector, not f(x, y).

```r
# define function to optimize
g <- function (v) {
  x <- v[1];
  y <- v[2]
  g<- x^2 + y^2
}
```

```r
## Optimize function
optim(c(1, 1), g)
```

```
## $par
## [1] 3.754010e-05 5.179101e-05
##
## $value
## [1] 4.091568e-09
##
## $counts
## function gradient
##       63       NA
##
## $convergence
## [1] 0
##
## $message
## NULL
```

Packages

The open-source nature of R encourages other people to write their own functions for their particular data-type or analyses.

Packages are college collection of objects (functions, or data) distributed through repositories. The most-common ones are CRAN and Bioconductor. CRAN alone has many thousands of packages.

The Packages tab in the bottom-right panel of R Studio lists all packages that you currently have installed. Clicking on a package name will show a list of functions that available once that package has been loaded.

Install packages

There are commands for installing packages within R. If your package is part of the main CRAN repository, you can use install.packages. We will be using the dplyr R package in this practical. In fact:

```
## the function dplyr is a collection of functions for data manipulation
##install.packages("dplyr")
## the ggplot2 is a collection of functions for data visualization
##install.packages("ggplot2")
## the readr is a collection of function for importing data
##install.packages("readr")
## the tidyverse is a collection of function for plotting data
##install.packages("tidyverse")
```

Once a package is installed, the library function is used to load a package and make it's functions (or data) available in your current session. You need to do this every time you load a new RStudio session:

```
## load the package dplyr
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
```

```
## load the package ggplot2
library(ggplot2)
## load the package readr
library(readr)
## load the package tidyverse
library(tidyverse)
```

```
## -- Attaching packages --------------------------------------- tidyverse 1.3.0 --
```

```
## v tibble  3.0.4     v stringr 1.4.0
## v tidyr   1.1.2     v forcats 0.5.0
## v purrr   0.3.4
```

```
## -- Conflicts ------------------------------------------ tidyverse_conflicts() --
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
```

Help for packages

A package can provide a lot of new functions. You can read up on a package on it's CRAN page, but you can also get help for the package within R using the library function.

```
# help for the dplyr package
library(help="dplyr")
```

Data manipulation

Import data

There is a series of R base functions read( ) that can help us to import data of different format. Any comma-separated-value (.csv) file can be imported using the function read_csv( ). All we need is to specify the path where the file is stored and assigning it to a new object to store the result.

A useful sanity check before importing the data is to use the function file.exists ( ) which will print TRUE is the file can be found in the working directory.

```
# check if data exists
file.exists("GDP_205U_NOC_NB_A-filtered-2020-12-22.csv")
```

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

After confirmed the .cvs data is in the path, we can import the data, using the option header = "" to define whether the in first row are stored variable names (TRUE) or data (FALSE).

```
# import data on GDP per capita across countries
data_gdp <- read.csv("GDP_205U_NOC_NB_A-filtered-2020-12-22.csv",  header = TRUE)
# rename columns
names(data_gdp) <- c("country", "label", "source", "year", "gdp")
```

Subset data

Selecting columns We can access the columns of a data frame using the select function. For instance, we can select column by name, by adding bare column names after the name of the data frame, separated by a comma.

```
## Select column country
data_country<-select(data_gdp, country)
```

while you can select a range of column by using two dots:

```
## Select columns country to year
data_country_year<-select(data_gdp, country:year)
```

We can also remove columns from by assigning a NULL to that column

```
## Remove column label
data_gdp$label <- NULL
## Remove column source
data_gdp$source <- NULL
```

Restricting rows To filter the rows in a data frame we can use a logical test. This logical test will be applied to each row and give either a TRUE or FALSE result. When filtering, only rows with a TRUE result get returned.

```
## Select rows of selected year
data_gdp_2018 <- data_gdp[data_gdp$year == "2018",]
```

```
## Select row for selected country
data_gdp_uk <- data_gdp[data_gdp$country == "United Kingdom",]
```

Sorting

To sort a data frame in R, use the order( ) function. By default, sorting is ASCENDING. If we add the sign minus before the sorting variable we indicate DESCENDING order:

```
# sort by gdp values in ascending order
data_gdp_2018_sorted <- data_gdp_2018[order(data_gdp_2018$gdp),]
# sort by gdp values in descending order
data_gdp_2018_sorted <- data_gdp_2018[order(-data_gdp_2018$gdp),]
```

Merge data

If we want to merge two datasets by some charachteristics, we can use the command merge( ) and specify the characteristics by which we merge the data using the option by=c( ).

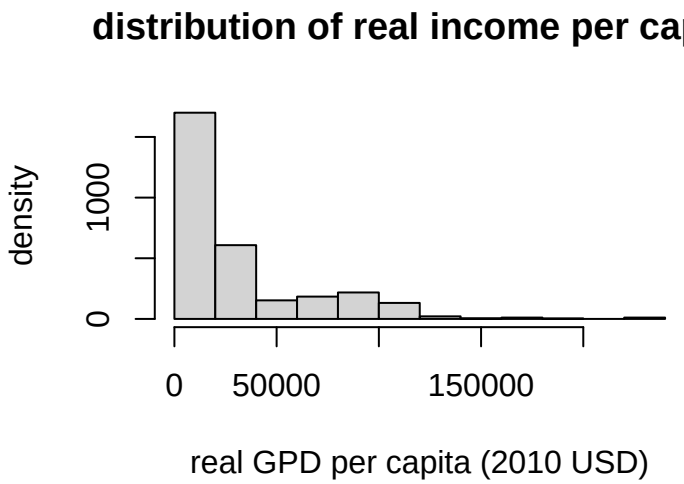
```
# import new data:hours worked rate across countries
data_hours <- read.csv("HOW_TEMP_SEX_ECO_GEO_NB_A-filtered-2021-01-12.csv",  header = TRUE)
# rename columns
names(data_hours) <- c("country", "label", "source",
                       "gender","occupations","area",
                       "year", "weeklyhours", "status",
                       "note1", "note2")
# subset matrix according to gender groups: Sex=all gender
data_hours<-data_hours[data_hours$gender == "Sex: Total",]
# subset matrix according to occupations groups: occupations=Aggregate
data_hours<-data_hours[data_hours$occupations == "Economic activity (Aggregate): Total", ]
# subset matrix according to area groups: area=Aggregate
data_hours<-data_hours[data_hours$area == "Area type: National", ]
# merge two data frames by country name and year
data_final <- merge(data_gdp,data_hours,by=c("country","year"))
```

Data visualization

Histogram

Creating histograms is easy using the R base function hist( ).

```
# Create histogram of real GDP per capita
hist(data_gdp$gdp,
     xlab="real GPD per capita (2010 USD)",
     ylab="density",
     main="distribution of real income per capita")
```

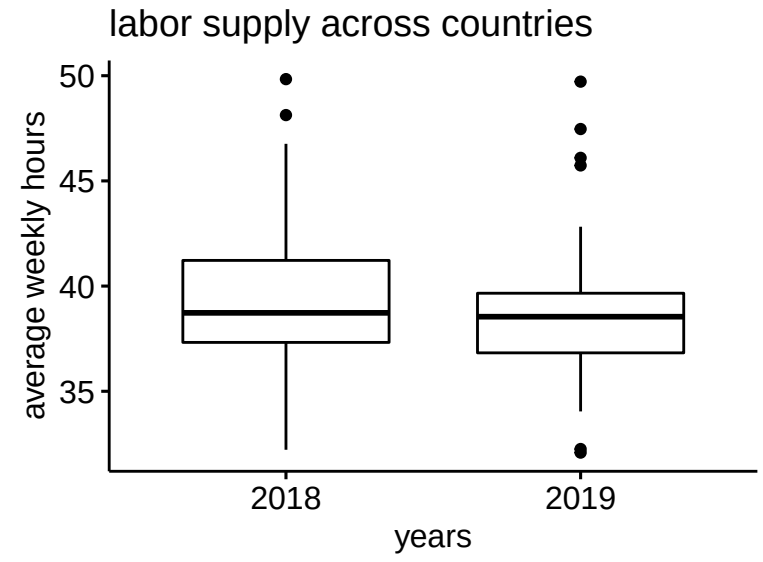


The option xlab="" allows to give a label to the x-axis while the option main="" allows to put a title to the plot.

Boxplot

We can create a boxplot using the function ggboxplot( ).

```
library("ggpubr")
# Select data for 2018 and 2019
data_hours_2018_2019 <- data_hours[data_hours$year >= "2018",]
# Create boxplot of hours worked by year
ggboxplot(data_hours_2018_2019, x="year", y="weeklyhours",
          xlab="years", ylab="average weekly hours", main="labor supply across countries")
```

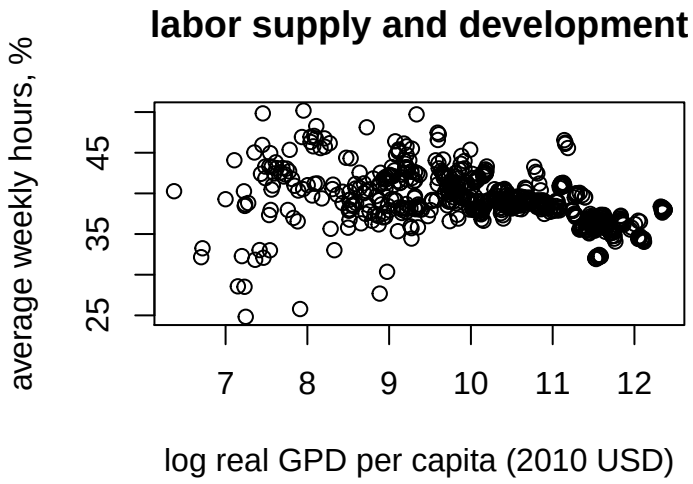


Scatterplot

Creating scatterplot is easy using the R base function plot( ).

```
# Create scatter of log GDP per capita and weekly hours
 plot(log(data_final$gdp),
```

```
      data_final$weeklyhours,
      xlab="log real GPD per capita (2010 USD)",
      ylab="average weekly hours, %",
      main="labor supply and development")
```



Another way of displaying these two variables is to use the function ggplot( ) which is part of the package called "tidyverse". Hence, we can write

```
# Create scatter of log GDP per capita and weekly hours
ggplot(data_final, aes(x = log(gdp), y = weeklyhours)) +
  geom_point() +
  stat_smooth( )  + # To fit a line, use stat_smooth(method='lm')
  ggtitle("labor supply and development") + # for the main title
  xlab("log real GPD per capita (2010 USD)") + # for the x axis label
  ylab("average weekly hours") # for the y axis label
```

```
## `geom_smooth()` using method = 'loess' and formula 'y ~ x'
```