Example notebook: iris dataset

Víctor H Cervantes

This is an example notebook using the dataset `iris` to illustrate literate programming. Here, we will also see examples of style and programming practices.

Preamble

Every script should start with general information about it script, such as:

- Name of the script
- Overview of what the script does/contains
- Description of how to use the script
- Author(s)
- Date

This list is not meant to be exhaustive.

The following chunk will contain this information for the script extracted from this notebook. It is always a good idea to name code chunks.

```
# iris_example.R ----
# This script loads the iris dataset (available within R),
# runs some basic sanity check of the data frame, and
# produces a few descriptive figures.
#
# Víctor H Cervantes (2023)
```

Additionally, at the beginning of each script, one should set up the environment[1]. This implies loading all packages needed and setting up general variables.

The following chunk gives an example for loading packages

[1]Not used in the technical R meaning here.

```
## Load packages ----
library(stats) # Actually stats is always loaded when R starts because it contains the most b
library(rprojroot) # This package can be used to help with paths
library(here) # This package can also be used to help with paths
```

```
here() starts at /home/herulor/ac/UIUC/2023-03_-_Fall/PSYC593/in_class_exercises/01_-_organiz
```

```
              # The package description says this one is meant to be used in interactive mode
              # Note that when a comment is long, it helps to break it and align it.
```

In the code above, note that the first line is a comment (it starts with #) that ends with four dashes (----). When a comment is written like this, RStudio identifies it as a section in the code. The hierarchy of these sections is determined by the number of #s, just like the `markdown` syntax.

By default, RStudio uses the project location (the my__project.Rproj file that it creates) as the working directory. However, it will not hurt to make sure. Specially because sometimes that working directory may not match (see more information at rprojroot website). So, we will set up some variables to deal with the paths later on:

```
## Global variables ----

### Path variables ----
# Root directory: the location from where R runs and looks for stuff
#                 aka as the working directory.
# Each assignment uses an alternative method
root_path <- rprojroot::is_rstudio_project        # Using rprojroot knowing this is an RStudi
here_path <- here::here()                         # Just using here. It follows additional he
                                                  # to try to identify where the root directo
crit_path <- rprojroot::find_root(has_dir("src")) # Using the directory structure we defined

# Some of the subdirectories
code_path <- file.path(root_path, "src") # Using the base R function file.path
docs_path <- here::here("doc")           # Using here
data_path <- file.path(here_path, "data")
figs_path <- file.path(crit_path, "results", "figures")
```

Using either `rprojroot` or `here` entirely avoids the issues associated setting the working directory. In particular, it saves you from setting the working directory by hand. Moreover, you should *never* **ever** *ever* use

```r
# Note that this chunk does not go into the script.
# This is done setting the option purl = FALSE.
# Also, the code does not actually run (which would produce an error).
# The option eval = FALSE achieves this.
setwd("path_to_where_I_have_my_stuff/but_not_to_where_you_have_yours")
```

The iris data

The text file we started with, is simply the R documentation for the dataset. It can be accessed by `help(iris)`.

> This famous (Fisher's or Anderson's) iris data set gives the measurements in centimeters of the variables sepal length and width and petal length and width, respectively, for 50 flowers from each of 3 species of iris.
>
> The species are Iris setosa, versicolor, and virginica.

Read the data

First, we need to get the data. The dataset is located in the `data/raw_data` subfolder.

```r
## Load the dataset ----
# From the csv provided
iris_data <- read.csv(file = file.path(data_path, "raw_data", "iris.csv"))

# From the data already available in R
data(iris)
```

Then, we check that the data was read correctly. This is usually checked looking at the first rows of the `data.frame` where the data is stored:

```r
## Check first rows

## read from file
head(iris_data)
```

```
  X Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1 1          5.1         3.5          1.4         0.2  setosa
2 2          4.9         3.0          1.4         0.2  setosa
3 3          4.7         3.2          1.3         0.2  setosa
```

```
4 4          4.6         3.1          1.5         0.2  setosa
5 5          5.0         3.6          1.4         0.2  setosa
6 6          5.4         3.9          1.7         0.4  setosa
```

```
## R's dataset
head(iris)
```

```
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1          5.1         3.5          1.4         0.2  setosa
2          4.9         3.0          1.4         0.2  setosa
3          4.7         3.2          1.3         0.2  setosa
4          4.6         3.1          1.5         0.2  setosa
5          5.0         3.6          1.4         0.2  setosa
6          5.4         3.9          1.7         0.4  setosa
```

Except for the additional column of row identifiers, both look the same. Also note that this kind of check is important while the script is being created (and also when data has changed), but may be omitted when everything is settled. Hence, this chunk will not go into the extracted script. The remaining of this script will mostly look into the original data.frame provided with R.

Just looking at the first rows of the dataset, however, is rarely effective for identifying problems loading the data. For example, knowing if not all variables and cases were read may be a better signal of problems somewhere (usually in the arguments used or missing from the function call used to load it).

```
## Check dimensions
(iris_dimensions <- dim(iris))
```

```
[1] 150   5
```

From the documentation, we know that

> iris is a data frame with 150 cases (rows) and 5 variables (columns) named Sepal.Length, Sepal.Width, Petal.Length, Petal.Width, and Species.

and we verify that indeed there are 150 rows and 5 columns.

We can also verify the column names in our data.frame

```
# Check column names of the data.frame iris
names(iris)
```

```
[1] "Sepal.Length" "Sepal.Width"  "Petal.Length" "Petal.Width"  "Species"
```

Yet, even when everything has been read in a table with the correct dimensions, there are times when the data has not been read correctly. A relatively easy way to check this, and also start exploring the data is to summarize the data set with the function `summary`

```
# Summary of the data depending on the type of the data
summary(iris)
```

```
  Sepal.Length    Sepal.Width     Petal.Length    Petal.Width   
 Min.   :4.300   Min.   :2.000   Min.   :1.000   Min.   :0.100  
 1st Qu.:5.100   1st Qu.:2.800   1st Qu.:1.600   1st Qu.:0.300  
 Median :5.800   Median :3.000   Median :4.350   Median :1.300  
 Mean   :5.843   Mean   :3.057   Mean   :3.758   Mean   :1.199  
 3rd Qu.:6.400   3rd Qu.:3.300   3rd Qu.:5.100   3rd Qu.:1.800  
 Max.   :7.900   Max.   :4.400   Max.   :6.900   Max.   :2.500  
       Species  
 setosa    :50  
 versicolor:50  
 virginica :50  
```

Most frequent issues detected this way are apparent when a (truncated) frequency table is shown instead of the six-number summary. That happens when for a variable that is numeric has been read as a character vector or as a factor.

```
# Summary of the data depending on the type of the data
# for the data that was read from the file
summary(iris_data)
```

```
       X           Sepal.Length    Sepal.Width     Petal.Length  
 Min.   :  1.00   Min.   :4.300   Min.   :2.000   Min.   :1.000  
 1st Qu.: 38.25   1st Qu.:5.100   1st Qu.:2.800   1st Qu.:1.600  
 Median : 75.50   Median :5.800   Median :3.000   Median :4.350  
 Mean   : 75.50   Mean   :5.843   Mean   :3.057   Mean   :3.758  
 3rd Qu.:112.75   3rd Qu.:6.400   3rd Qu.:3.300   3rd Qu.:5.100  
 Max.   :150.00   Max.   :7.900   Max.   :4.400   Max.   :6.900  
  Petal.Width      Species         
 Min.   :0.100   Length:150        
 1st Qu.:0.300   Class :character
 Median :1.300   Mode  :character
 Mean   :1.199
 3rd Qu.:1.800
 Max.   :2.500
```

For example, note that the data from the csv file has the variable `Species` as character, whereas the data in R has it as a factor. This difference was not visible at all from looking at the first rows of each data.frame.

A useful function that allows to gather most, if not all, of this information is `str`. It also shows explicitly the type of each of the variables, which would have been guessed out of the output from the previous functions.

```r
# The compact summary of the data.frame
str(iris)
```

```
'data.frame':   150 obs. of  5 variables:
 $ Sepal.Length: num  5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
 $ Sepal.Width : num  3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...
 $ Petal.Length: num  1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ...
 $ Petal.Width : num  0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ...
 $ Species     : Factor w/ 3 levels "setosa","versicolor",..: 1 1 1 1 1 1 1 1 1 1 ...
```

A few descriptive figures

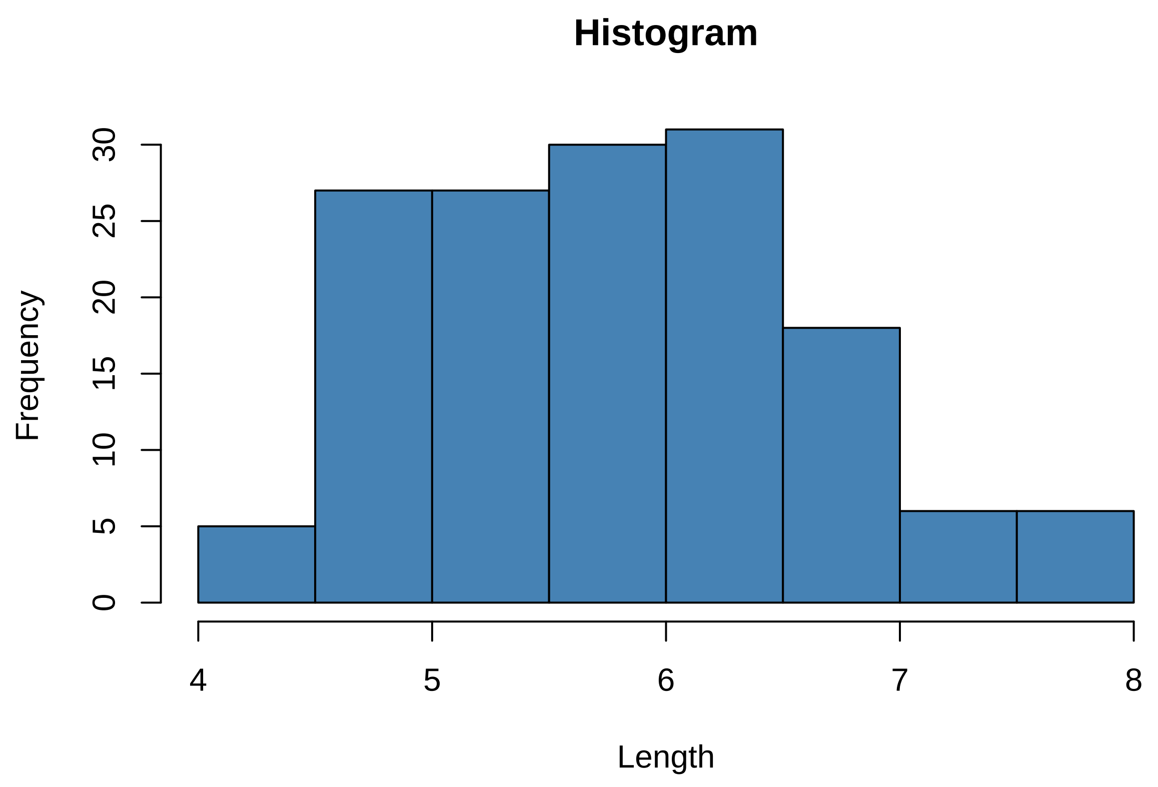
We would like to obtain a few descriptive figures from this data set and save them in the `results/figure` folder in our project.

We have three base R figures to include in this notebook:

1. A histogram,
2. a scatterplot, and
3. a grouped boxplot.

One alternative to include the figures in the document and make sure they are saved in the proper folder is:

```r
# Create a histogram
hist(iris$Sepal.Length,
     col  = 'steelblue',
     main = 'Histogram',
     xlab = 'Length',
     ylab = 'Frequency')
```



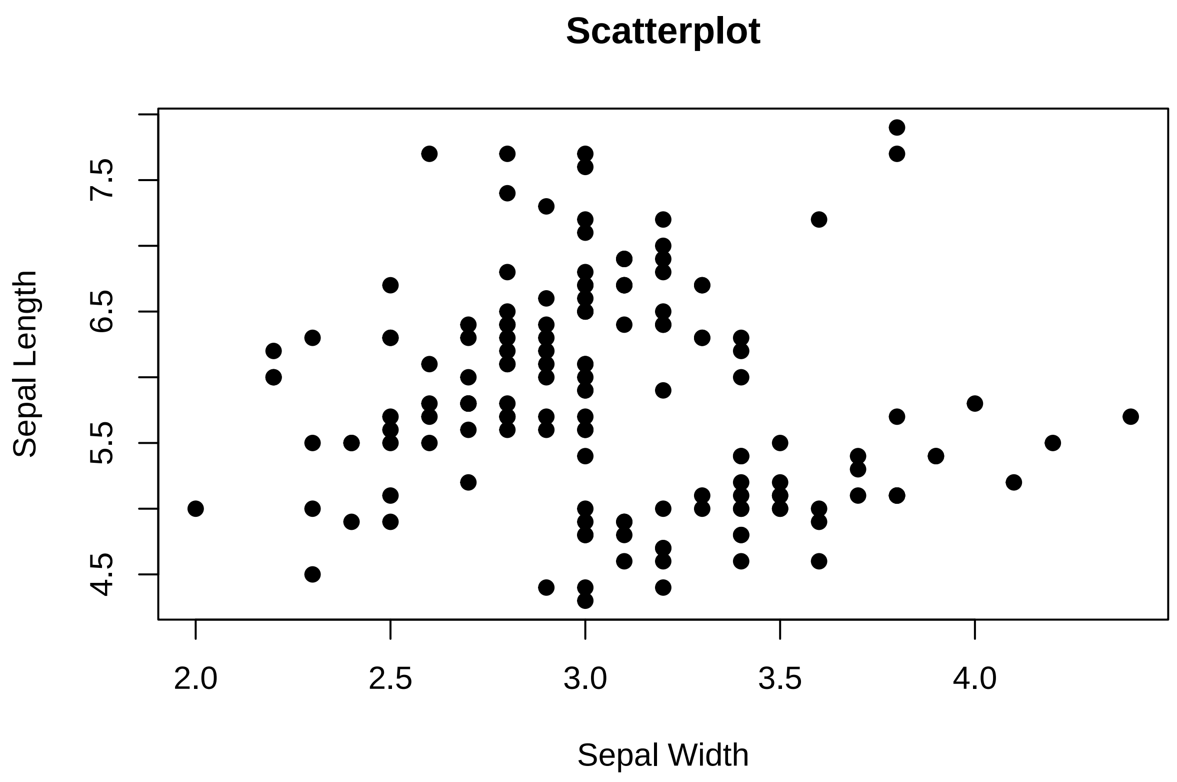
```
# Save a histogram
png(filename = file.path(figs_path, "histogram.png"))
hist(iris$Sepal.Length,
     col  = 'steelblue',
     main = 'Histogram',
     xlab = 'Length',
     ylab = 'Frequency')
dev.off()
```

```
pdf
  2
```

One should be able to achieve this without repeating code… but at least at this attempt on this document, I was forced to repeat it to get both the figure saved and in the compiled file.

A preferable way to get the file saved is to use the option `fig.path` on the chunk

```
# Create and save a scatterplot
plot(x     = iris$Sepal.Width,
     y     = iris$Sepal.Length,
     main = 'Scatterplot',
     xlab = 'Sepal Width',
     ylab = 'Sepal Length',
     pch  = 19)
```



Or even better it can be set globally to the options for all chunks, wither within a chunk or on the document header.

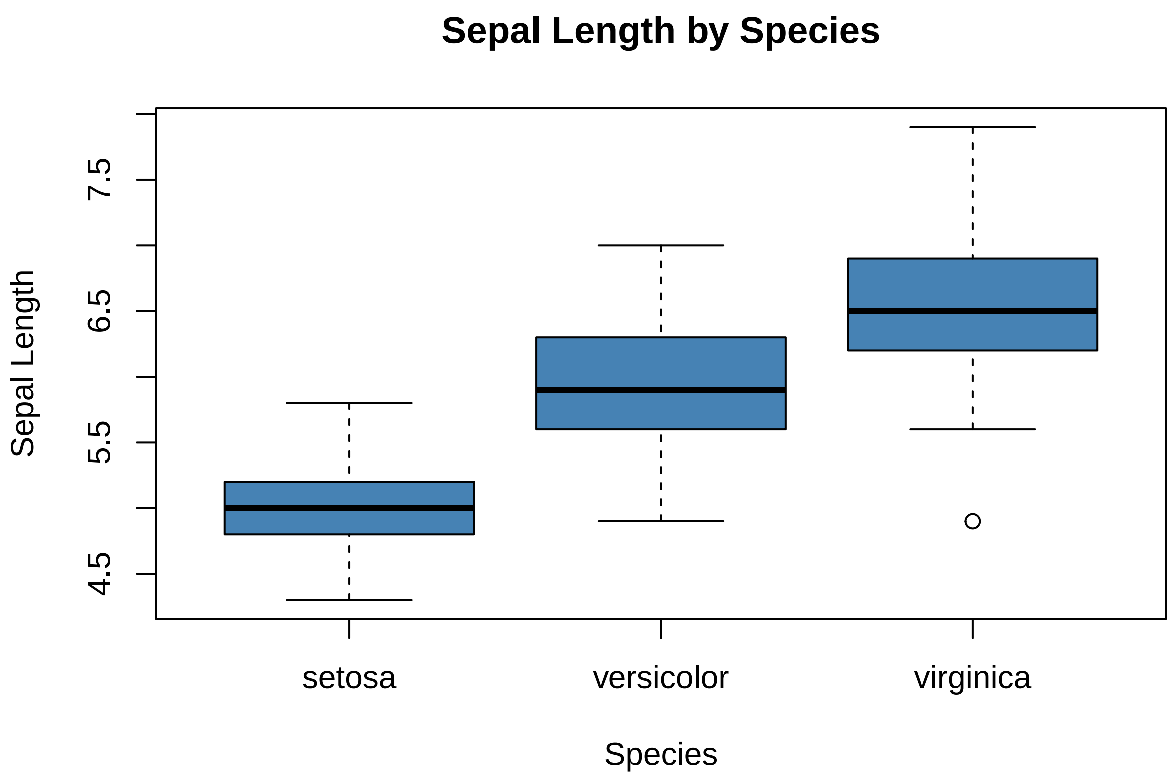
```
# Within a chunk
knitr::opts_chunk$set(fig.path = paste0(figs_path, "/"))
```

```
# as options to the document header
knitr:
  opts_chunk:
    fig.path: directory/to/store/figures/
```

Setting the options within a chunk will only change them from that point forward in the document. That is, it would be best to include that among the other global variables. When stated in the document header, it will apply throughout the whole document.

So now, we can generate the last plot: a boxplot grouped by Species.

```r
boxplot(Sepal.Length ~ Species,
        data = iris,
        main = 'Sepal Length by Species',
        xlab = 'Species',
        ylab = 'Sepal Length',
        col = 'steelblue',
        border = 'black')
```


Sepal Length by Species
Sepal Length
setosa
versicolor
virginica
Species