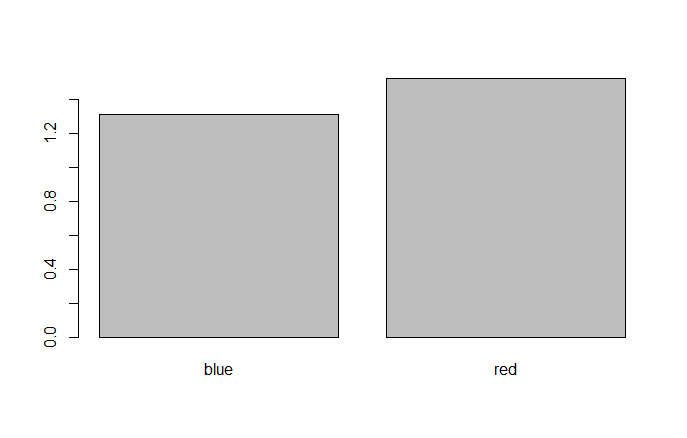
# Part III: Data manipulation and plotting

In part three we focus on workflow operations 3a (data manipulation) and 5a (plotting a high-quality graph) and 5b (getting descriptive statistics out of R). In the previous parts, the help page was printed as an aid in various situations. From here on, this information is shown less frequently. While working through the script, it is nevertheless a good idea to check the help pages.

## Chapter 8. First “pretty plot” of analysis

Producing high-quality graphs of your data is a major incentive to stick to the R workflow (Figure 5.0). One such workflow includes (operation 1) reading in the data, (operation 2) checking the data, (operation 3) filtering of data, (operation 4) calculating statistics from these data. The basics of these steps in the workflow were introduced in the previous chapters. We here focus on plotting the descriptive statistics in a high-quality plot (operation 5a).

### 8.1 barplot()

Figure 8.1.1 Mean mass of blue and red individuals plotted using the default setting of barplot()

To illustrate the plotting operation, the function barplot() is used, but as we will see in further chapters many of the coding translates to other plotting functions. As seen from its help page (Box 8.1), the minimal requirement for this function is information on the height of the bars to plot. We can, e.g. readily plot the means of differently coloured individuals by using the information on mean mass calculated using tapply() in section 7.2 above,

> barplot(means.per.colour)

which produces a fairly dull looking barplot (Fig. 8.1). Most importantly, the barplot lacks a label for the axes, although it has named the bars correctly. Study of the help page shows that to specify the axes labels xlab and ylab are used for X and Y axes respectively. It may also be good to increase the font size of the names and numbering of the axes (cex.axis, cex.names). The argument cex works as a multiplication factor of the default font size such that specification of cex>1 increases the font size. By running

> barplot(means.per.colour,xlab=list(“Colour”,cex=1.5), ylab = list(“Mass (g)”,cex=1.5),cex.names=1.2,cex=1.2)

a better plot is obtained (Figure 8.1.2). In the above line of code, we specify (using cex) that the font size of the labels of the X and Y axes are to be increased by 50% (cex=1.5, i.e. 1.5 times the default font size). Note that the cex argument is provided together with the labels as an object of class list. We will look at list in more detail later on, but for now it is sufficient to realize that this list object contains both the name of the axes as a string, and the cex scaling parameter

> list(“Colour”, cex=1.5)

[[1]]

[1] “Colour”

$cex

[1] 1.5

*Box 8.1. Help page for barplot (shortened)*

|  |  |
| --- | --- |
| barplot {graphics} | R Documentation |

**Bar Plots**

**Description**

Creates a bar plot with vertical or horizontal bars.

**Usage**

barplot(height, ...)

## Default S3 method:

barplot(height, width = 1, space = NULL,

names.arg = NULL, legend.text = NULL, beside = FALSE,

horiz = FALSE, density = NULL, angle = 45,

col = NULL, border = par("fg"),

main = NULL, sub = NULL, xlab = NULL, ylab = NULL,

xlim = NULL, ylim = NULL, xpd = TRUE, log = "",

axes = TRUE, axisnames = TRUE,

cex.axis = par("cex.axis"), cex.names = par("cex.axis"),

inside = TRUE, plot = TRUE, axis.lty = 0, offset = 0,

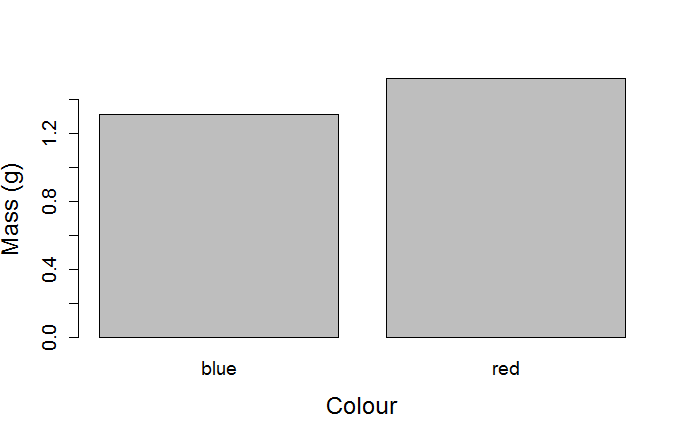
add = FALSE, args.legend = NULL, ...)

**Arguments**

|  |  |
| --- | --- |
| height | either a vector or matrix of values describing the bars which make up the plot. If height is a vector, the plot consists of a sequence of rectangular bars with heights given by the values in the vector. If height is a matrix and beside is FALSE then each bar of the plot corresponds to a column of height, with the values in the column giving the heights of stacked sub-bars making up the bar. If height is a matrix and beside is TRUE, then the values in each column are juxtaposed rather than stacked. |
| width | optional vector of bar widths. Re-cycled to length the number of bars drawn. Specifying a single value will have no visible effect unless xlim is specified. |

----

The resulting bar plot (Figure 8.1.2) is now intelligible, but still looks quite grey. We can introduce color into the plot by the argument col. We also computed more descriptive statistics; the standard deviation of differently colored individuals as well as the sample sizes of these two subsets. We can add these pieces of information by using two additional plotting functions, segments() and text(). These functions place a line, respectively text in the X-Y plotting space defined by barplot(). However, the X-Y mapping is not clear from the figure: While we can more or less interpret the numbering of the Y-axis, we do not see the numbering of the X-axis. We can extract that information if we assign the barplot() to an object. The following line

*Figure 8.1.2: Barplot of the mean mass of differently coloured individuals produced by overriding the default settings of* barplot()

> barplot.X<- barplot(means.per.colour,xlab=list(“Colour”,cex=1.5), ylab = list(“Mass (g)”,cex=1.5),cex.names=1.2,cex=1.2)

> barplot.X

[,1]

[1,] 0.7

[2,] 1.9

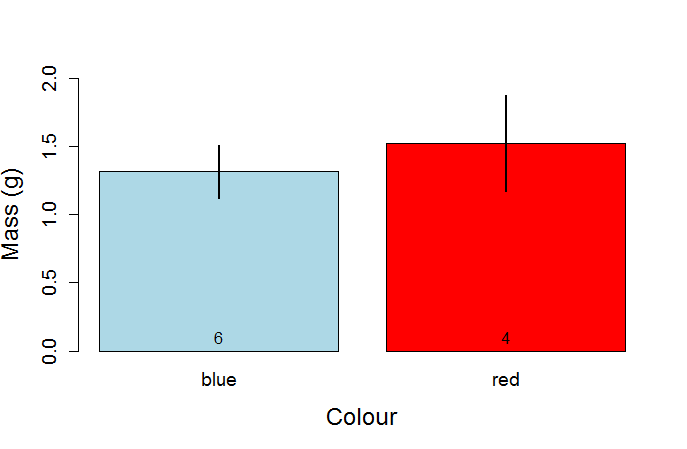
produces the barplot (Figure 8.1.2), and also an object with the X coordinates of the middle of the bars plotted. That is, the middle of the bar for “blue” is plotted at X = 0.7, with the height of the bar being 1.32, i.e. the mean mass of blue individuals. To show the extent of the SD in relation to the mean, a line can be plotted using segments() through the middle of the bar going from the mean mass minus 1 SD to the mean mass + 1 SD for each subset. We increase the width of this line by passing the argument lwd=2 to segments(). We write the sample size in the bar using text(). Because the mean + 1 SD exceeds the maximal value of the Y axis as automatically determined by barplot(), new limits of the Y axis need to be provided using the argument ylim to barplot(). While we are at it, we also add some color using the argument col to barplot(). The drawing of the “pimped” barplot (Figure 8.1.3) now takes three lines of coding

> barplot.X<-barplot(means.per.colour,xlab=list(“Colour”,cex=1.5), ylab = list(“Mass (g)”,cex=1.5),cex.names=1.2,cex=1.2,ylim=c(0,2), col=c(“lightblue”,”red”))

> segments(barplot.X,means.per.colour- sd.per.colour, barplot.X, means.per.colour+sd.per.colour,lwd=2)

> text(barplot.X,0.1,n.per.colour)

Take some time to study what is written in the previous paragraph and relating that to the above script. We have now coded the plotting of a nice-looking graph. In particular, note that the plot was constructed in steps; first the barplot was created, and then lines and text were added to the X-Y map that the call to barplot() created. This way of constructing a plot is a general aspect of plotting in R. Further adjustments can of course be made to the barplot as we have by no means introduced all possible modifications.

*Figure 8.1.3: Barplot of mean mass of blue and red individuals with the standard deviation (+/- 1SD) indicated by the line and sample size of each subset printed above the X axis*

We have now performed an entire workflow, from reading and checking the data (Part I) to the calculation of the descriptive statistics (Part II) to plotting these in high-quality graphs (this chapter). All these operations were done in R. While “coding a plot” may seem cumbersome to you, and likely represent initial investment in learning, remember that the coding is general. It will work, with no or minor adjustments, for other data to produce similarly good-quality figures. The use of a program dedicated to drawing figures will typically require also input by the user, quite often much “manual” input involving mouse clicking which does not leave a trace and therefore has to be done again when making new plots. By “coding a plot” you have the script, a reproducible record of what was done to arrive at the plot you wanted. Ready to be used again, if needed.

### 8.2 plot()

The function plot() was introduced in 4.4 as a quick graphical check or exploration of your data. Here we make the plot look nicer. A key aspect is that many of the aspects introduced in the previous section also apply to plot().

As in the barplot() example, we want to use colours to separate the subsets. This was straightforward for the barplot as there were two bars and two colours. For plotting each data point, however, we need a vector containing the colour to use for each data point. In this case, we of course have this information in myData, the vector myData$colour, but in general we may not have written the subsets as colours (e.g. groups are coded as “control”, “treatment”) or we may want to use different colours. We can make use of the fact that if we specify that colour is a factor, this will be internally coded using 1’s and 2’s. Below, a new column is added where colour is coded as a factor.

> myData$colourF<-as.factor(myData$colour)

> levels(myData$colourF)

[1] “blue” “red”

Thus, blue individuals are coded as 1 and red individuals as 2. Remember that R will code the factors in alphabetic order (see 3.4) so this coding is as it should be. The vector

> myData$colourF

[1] blue blue red blue blue red red blue blue red

Levels: blue red

therefore becomes a vector of 1’s and 2’s when interpreted as a number

> as.numeric(myData$colourF)

[1] 1 1 2 1 1 2 2 1 1 2

Because of this, we can construct a vector cl of the colours we want to use and use myData$colourF for indexing this vector, where each element in cl is the colour we want to plot each point.

> cl<-c(“lightblue”,”red”)

> cl[myData$colourF]

[1] “lightblue” “lightblue” “red” “lightblue” “lightblue” “red” “red” “lightblue”

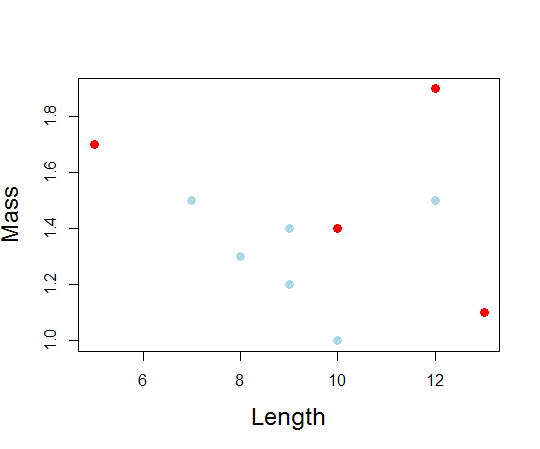
[9] “lightblue” “red”

Note in the above lines of code that we do not need to coerce the factorial vector myData$colourF to a numerical one, but R automatically interprets this vector as 1’s and 2’s.

We then include cl[myData$colour] as the argument col to plot().

> plot(myData$mass~myData$length,col=cl[myData$colourF],pch=19, cex=1.3, xlab=list(“Length”,cex=1.5), ylab=list(“Mass”,cex=1.5))

The above produces a plot of the two subsets distinguished by colour (Figure 8.2). It uses the pch argument to draw filled circles and the cex argument to increase the size of the filled circles. As before, xlab and ylab arguments define the labels and their font size.

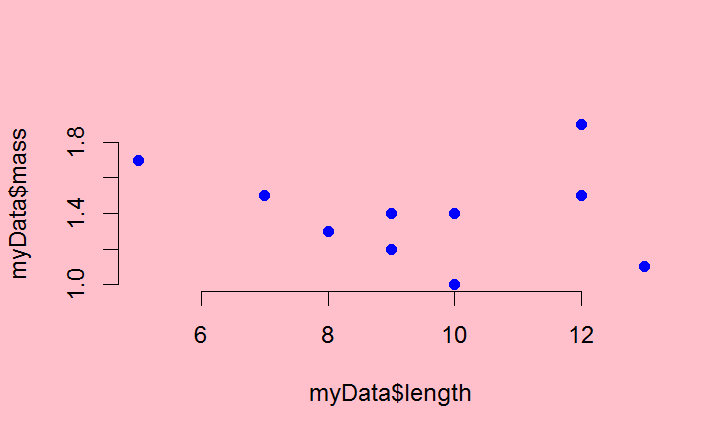
*Figure 8.2: Mass plotted against length for individuals of blue and red colour as indicated by the respective coloration of the plotted dots.*

### 8.3 Graphical parameters: find your way in the maze of options

Plots in R are highly adjustable. The help page for par() contains lots of information that can help you improve the quality of your plots considerably. The function par() itself is used to set graphical parameters, and there are many! The idea is that by setting these graphical parameters you can personalize (i.e. change the default) of plots. For example, I can decide I want all my plots to have increased font size by 50%, a filled dot, a pink background, plot in blue and not draw the square box around the plot. I can set these graphical parameters and the same plot() command for my “quick and dirty” plot (Figure 4.4) will produce a very different figure (Figure 8.3).

> par(cex=1.5, pch=19, bg=”pink”, col=”blue”,bty=”n”)

> plot(myData$mass~myData$length)

*Figure 8.3: Quick graphical check plot of chapter 4.4 drawn after overriding the default graphical parameters using* par()

Because I have now set these parameters, they will affect the way all my plots will look. For example, also my quick boxplot() (Figure 4.3) will now be plotted on a pink background. This may not be what you want. It is typically more useful to specify (most) graphical parameters when executing the plot command. Luckily, almost all graphical parameters in this long list can be used directly in the various plotting functions and you do not need to specify them prior to plotting using par(). Nevertheless, you can find information on how to “pimp your plot” from the help page of par(). For example, the arguments cex, pch, lwd used above are all in the help on par(). Remember that you can in R Studio also search within the help page (“Find Topic”). For example, you can search for “dashed” on the help page of par() to find out how to create a dashed line. The R Reference Sheet is also helpful for this purpose.

Similarly, the help page for the function points() has an overview of how you can change the default open circle to various symbols (argument pch). This function allows you to overlay a second plot on top of a first plot. Again, even when you do not use this function, you can use the pch argument in plot() as above.

The function colours() provides a list of all named colours when you run this command. Using a named colour is the easiest way to modify the colour in your plot. Note that multiple colours have a number after their name (e.g. “red”, “red2”, “red3”, …), where higher numbers mean a darker colour. In contrast, the grey scale works such that higher numbers refer to lighter grey (“grey80” is lighter than “grey20”). The help page for this function (or the “Color Specification” section at the bottom of the help for par() details how you can obtain even more options.

As you become more familiar with working in R, you will quickly become more accustomed to the many graphical parameters and various options available.

### 8.4 Exporting figures from R Studio

We have earlier (4.5) encountered the built-in options for exporting plots in R Studio. You can also use the function pdf() to generate a PDF file of your figure. The PDF format is highly flexible (see its help page), and works across platforms. You can save Fig. 8.2 to your working directory as a PDF file

> pdf(file=”Fig8\_3.pdf”,paper=”a4”)

> plot(myData$mass~myData$length,col=cl[myData$colour],pch=19, cex=1.3, xlab=list(“Length”,cex=1.5), ylab=list(“Mass”,cex=1.5))

> dev.off()

null device

1

The call to function pdf() opens a named file (or creates this file if it does not exist) in your working directory. The argument paper specifies what kind of paper size you use (handy if you want to print it; here “a4”, but “letter” could be used in USA locale). You then execute the plot() code (can be multiple lines of code) that creates your plot. You must end with the function dev.off() which terminates the connection with the file (without this command, you cannot open the file).

## Chapter 9. Coding an own function

### 9.1.1 function()

A function carries out a specific set of instructions and returns the result. For example, sum() and colSum() are functions (to sum all up all elements or to sum up only part of the elements, respectively). You can program a function yourself in R using function(). For example, you can write a function to Z-standardize a data vector called Zstand. For a vector, the Z-value is the value of an element in that vector on the data scale minus the mean of all the values in the vector divided by the standard deviation of all the values in the vector. Z values are used whenever it is important to standardize your variable such that is has zero mean and its units are in terms of 1 standard deviation. In script notation

Zstand<-function(x) {

mean.x<-mean(x)

sd.x<-sd(x)

z.x<-(x-mean.x)/sd.x

return(z.x) #writing return is needed if an object is defined

}

where R Studio will automatically indent the code within curly brackets {…}. The script editor in R Studio will help you with the curly brackets. Firstly, for curly brackets a little “arrowhead” will appear in the margin where the line numbers are. At the line where the opening curly bracket ({) is the arrowhead will point down and at the line where the closing curly brackets (}) is the arrow head will point up. By clicking on these arrowhead you can collapse all the code that is between curly brackets. In addition, when you place your cursor on any kind of bracket, R Studio will highlight its corresponding bracket and this works also for curly brackets close } and open {.

Running the above snippet of code produces in the Console

> Zstand<-function(x) {

+ mean.x<-mean(x)

+ sd.x<-sd(x)

+ z.x<-(x-mean.x)/sd.x

+ return(z.x) #writing return is needed if an object is defined

+ }

> class(Zstand)

[1] “function”)

where the +’s indicate that the Console is waiting for further lines of code.

By the way, if you ever get stuck in this situation where R is waiting (showing +) whereas you expect it to be ready (showing >), move to the Console (place your cursor there and mouse click) and press “Esc” (escape) or press the “Stop” button in R Studio’s Console panel if available.

The code above starts by specifying we want to make an object called Zstand() which is a function of x (the input variable). What is written between the curly brackets ‘{ … }’ is a little program which specifies what Zstand() does, and is here ended with the function return() specifying what the object Zstand() will return when called; in this case z.x which is the Z-standardised value of x. For a vector v, calling Zstand(v) thus returns the Z-standardised values of v.

> Zstand(c(1,2,3,4))

[1] -1.1618950 -0.3872983 0.3872983 1.1618950

where the Z-standardised values are printed in the Console. For R to remember the values it should be assigned to an object.

> z.x-Zstand(c(1,2,3,4))

As indicated in the annotation above, we do not need to include a the return() function, which would make the script a bit shorter

Zstand<-function(x) {

mean.x<-mean(x)

sd.x<-sd(x)

(x-mean.x)/sd.x

}

### 9.1.2 Note on coding

You are now getting started with some nice coding, and a good rule is “if it works, it works”. Note, however, that the function Zstand() as defined above contains many redundant temporary vectors. Vectors mean.x and sd.x are defined explicitly, but used only once. The coding of this function can thus be reduced to be more compact by not writing out these “help” vectors. That is, we can equivalently code

Zstand<-function(x) {

z.x<-(x- mean(x))/ sd(x)

return(z.x)

}

which can be reduced even further by removal of return()

Zstand<-function(x) {

(x- mean(x))/ sd(x)

}

All these versions produce the same answer and are functionally equivalent. This situation is fairly typical when coding. The first version of Zstand() (in 9.1.1) has the advantage that it breaks down the code into its smallest possible steps: (1) calculate the mean, (2) calculate the standard deviation, (3) calculate the input vector minus mean and divide by standard deviation. The second and last version of Zstand() (in 9.1.2) performs these steps in one go. The last version is clearly very compact and still very intelligible given that we use functions with intuitive names like mean() and sd(). Compact coding, and especially avoiding to define redundant objects are important in advanced applications involving either many calculations and/or Big Data or similar. For many applications, however, the compactness of your code is largely irrelevant. When starting out, you should primarily strive to code in a way that is intelligible and natural to you, so you understand what your code does. As you proceed in coding your solutions, and gain confidence you likely find that your coding changes and may become more compact.

## 9.2 Default arguments in a function

To better understand how R uses default values in functions, it is instructive to make an own function which contains a default. The function below returns the value(s) given minus a desired value. We can when setting up the function create a default value to use as “minus value”, which we define as myMinusValue=1. This means that unless the user specifies another value, this default value will be used. The default is specified within function().

> minusValue<-function(x,myMinusValue=1) {

+ minus.x<-x-myMinusValue

+ return(minus.x)

+ }

Thus, if we do not specify the argument myMinusValue, our function will assume that myMinusValue=1.

> minusValue(2)

[1] 1

We have to override the default if we want the function minusValue() to use another value than 1. For example, myMinusValue could be 2.

> minusValue(2,2)

[1] 0

Another version of the above coded more compactly is

> minusValue<-function(x,myMinusValue=1) {

+ x-myMinusValue

+ }

## 9.3 Example: coding a function to calculate sample size with missing values

There are a number of handy basic functions to get at properties of your data. One such function is unique(), which works both for numbers and strings. It returns which entries in a vector are unique (i.e. it ignores the repeats). For example,

> vec<-c(1,2,2,3,3,4,4,5,5,6,7,7,8,8,9,10)

> unique(vec)

[1] 1 2 3 4 5 6 7 8 9 10

> vec.s<-c("a","b","b","c","b","a","c")

> unique(vec.s)

[1] “a” “b” “c”

You can count how many entries there are in a vector, by using length(). Thus,

> length(vec)

[1] 16

> length(unique(vec))

[1] 10

where the former gives the total number of elements in the vector and the latter the number without the repeats.

Note that the use of length() means you are counting the number of elements in a vector. As explained above, NA (missing values) are counted as elements (because they are!). Thus

> length(c(1,2,3))

[1] 3

> length(c(1,NA,3))

[1] 3

In the latter case, the sample size is clearly not 3 and thus using length() to obtain your sample size (as in chapter 7) produces erroneous results. When there are missing values, you need to remove these if you want to calculate sample size. Sadly, na.rm=TRUE which we encountered for the function mean() is not an argument for the function length(). There are multiple ways.

One function is na.omit() which will strip away the NA values

> length(na.omit(c(1,NA,3)))

[1] 2

Another approach is to use the fact that logical vectors (vectors consisting of TRUE and/or FALSE) when summed produce a sum of all the elements which are TRUE. That is, a logical vector is – from this perspective – viewed as a binary vector of 1 (TRUE) and 0 (FALSE). To get the logical vector saying whether each element is NA or not, a number of approaches can be used. For example, complete.cases() (which creates a logical vector stating whether there is an NA or not), or is.na() which creates a logical vector stating whether each element is NA (TRUE) or not (FALSE) which then of course needs to be reversed using !

> complete.cases(c(1,NA,3))

[1] TRUE FALSE TRUE

> sum(complete.cases(c(1,NA,3)))

[1] 2

> !is.na(c(1,NA,3))

[1] TRUE FALSE TRUE

> sum(!is.na(c(1,NA,3)))

[1] 2

Clearly, is.na() can be used to obtain the number of missing values

> sum(is.na(c(1,NA,3)))

[1] 1

We now have all the ingredients to code our own function to calculate sample size of any vector irrespectively of whether it holds NA or not. Such a function could look like

> sampleSize<-function(x) {

+ # function to calculate sample size stripping away missing values

+ sum(complete.cases(x))

+ } #function(x)

> sampleSize(c(1,2,NA,4))

[1] 3

To put the function into use we can consider the following data

> df.withNA<-data.frame(Trial=1:10, Experiment=c(rep(“control”,5),rep(“treatment”,5)),Measure=c(1,3,4,NA,3,4,5,NA,8,9))

> df.withNA

Trial Experiment Measure

1 1 control 1

2 2 control 3

3 3 control 4

4 4 control NA

5 5 control 3

6 6 treatment 4

7 7 treatment 5

8 8 treatment NA

9 9 treatment 8

10 10 treatment 9

where in two trials the measure failed resulting in a missing value (NA). To obtain the sample size per experimental subset, we use our new function within the tapply() function

> tapply(df.withNA$Measure,df.withNA$Experiment,sampleSize)

control treatment

4 4

This clearly counts the sample size correctly, as opposed to using length().

> tapply(df.withNA$Measure,df.withNA$Experiment,length)

control treatment

5 5

When dealing with missing values (NA), the details of the functions start to matter. For example, if we use aggregate(), we obtain the correct sample size (i.e. without NA) also when using length()

> aggregate(Measure~Experiment, data=df.withNA,sampleSize)

Experiment Measure

1 control 4

2 treatment 4

> aggregate(Measure~Experiment, data=df.withNA,length)

Experiment Measure

1 control 4

2 treatment 4

Remember from the R help for aggregate() (Box 7.1) that the above formulation is the so-called formula method of aggregate(). When used with the formula method, aggregate() will by default have the argument na.action=na.omit. Note that this is only the case if aggregate() is used with the formula method and that other methods will not strip NA values. For example

> aggregate(df.withNA$Measure,list(df.withNA$Experiment), length)

Group.1 x

1 control 5

2 treatment 5

## 9.4 Anonymous function

The functions aggregate() and tapply() (and other members of the apply() family) are great tools. These functions apply a function on a subset of your data. As we saw above, the user can define the function as a named object of class function. Alternatively, if the function is really short and simple, you can create the function within the data extraction function itself. It has, in that case, no name and is therefore considered an *anonymous function*. For example,

> tapply(df.withNA$Measure,df.withNA$Experiment,function(x) sum(complete.cases(x)))

control treatment

4 4

where instead of giving within tapply()the argument sampleSize(), this function is written out. In doing so, the curly brackets are not needed (but including them does no harm).

Anonymous functions are not necessary in the sense that you can achieve the above equally well by first specifying the function explicitly (i.e. assign it to an object of class function, and then use this function in the call to tapply()). Nevertheless, using anonymous functions allows for clearer coding (after some practice), as it is clear from the call to tapply() what is done without the need to go back in the script to see what sampleSize() does. In particular, anonymous functions are handy if two or perhaps three functions are to be combined. With two functions, sum() and complete.cases() we could in the above example obtain sample sizes excluding missing values. Another way is to combine the functions length() and na.omit() where the latter strips NA from an object.

> tapply(df.withNA$Measure,df.withNA$Experiment,function(x) length(na.omit(x)))

control treatment

4 4

And of course there are more options. In the end, it is a matter of taste and style how and which functions you implement in these kind of situations.

## Chapter 10. Saving your results

In this chapter we focus on workflow operations 5b. In Part II and in this Part, we have started exploring R’s powerful functions for applying functions over subsets (apply family) which can be used to compute descriptive statistics. We have also started with how to write purpose-specific functions to allow us to compute purpose-specific properties. So far, the resultant output has only been printed on the Console, or then in R’s brain, perhaps even as different objects. Just as we can export our plots, we likely also want to combine the descriptive stats, e.g. together nicely in a summary table and export this information. The two basic ways to convey information are, after all, figures and tables.

### 10.1 Table of descriptive statistics

Before doing more advanced data analysis, it is a good habit to explore your data. One aspect is to present descriptive statistics. As a simple approach, we can construct a data.frame which we fill with descriptive statistics on subsets of df.withNA. The function aggregate() is well suited for this purpose. Let’s start with the mean

> descriptiveStats.df.withNA<-aggregate(Measure~Experiment, data=df.withNA,mean)

> descriptiveStats.df.withNA

Experiment Measure

1 control 2.75

2 treatment 6.50

Notice that this call to aggregate() produces the mean measure despite there being NA values. The help page (Box 7.1) explains why: when using the formula method, aggregate() has as default additional argument na.action=na.omit. Intuitively, this default means that the NA values are omitted prior to applying the function specified to the different subsets (performed by the function na.omit(); see also 9.4). What is made here is a data.frame

> str(descriptiveStats.df.withNA)

'data.frame': 2 obs. of 2 variables:

$ Experiment: chr "control" "treatment"

$ Measure : num 2.75 6.5

Thus, we can add columns (see 5.2.4) to this data.frame with more descriptive statistics

> #add n

> descriptiveStats.df.withNA<-data.frame(descriptiveStats.df.withNA, n=aggregate(Measure~Experiment, data=df.withNA,length)$Measure)

> #add sd

> descriptiveStats.df.withNA<-data.frame(descriptiveStats.df.withNA, sd=aggregate(Measure~Experiment, data=df.withNA,sd)[,2])

> names(descriptiveStats.df.withNA)[2]<-“mean”

> descriptiveStats.df.withNA

Experiment mean n sd

1 control 2.75 4 1.258306

2 treatment 6.50 4 2.380476

Note that each call to aggregate() will create a data.frame with the same structure; it has two columns whose tags are $Experiment and $Measure. We only want to add the descriptive stats of the measure ($Measure or [,2] as illustrated in lines above for n and sd respectively). Clearly we can add additional descriptive stats following the same logic.

We have seen that aggregate() can apply a function to multiple variables (using cbind(), see Box 7.1). Now, you may wonder whether we can also execute multiple functions at the same time. To do so we have to – as in chapter 9 – code our own function. We can combine the computation of multiple descriptive statistics:

> descriptiveStats.df.withNA.multi<-aggregate(Measure~Experiment, data=df.withNA,function(x) c(mean(x),length(x), sd(x)))

> #what did we make?

> str(descriptiveStats.df.withNA.multi)

‘data.frame’: 2 obs. of 2 variables:

$ Experiment: Factor w/ 2 levels “control”,”treatment”: 1 2

$ Measure : num [1:2, 1:3] 2.75 6.5 4 4 1.26 ...

> descriptiveStats.df.withNA.multi

Experiment Measure.1 Measure.2 Measure.3

1 control 2.750000 4.000000 1.258306

2 treatment 6.500000 4.000000 2.380476

which clearly produces a rather odd object: A data.frame with its second “column” a 2 x 3 matrix which when printed to the Console, however, does produce the numbers we are after. While it is certainly possible to work further with this object (e.g. coerce the matrix to a data.frame) or with this approach in general (simply copy-paste the table from the Console), my preference is for the “line-by-line” option outlined above. It is tractable and also produces the correct tags.

### 10.2 Export the summary table as a text file

We read text formatted data using read.table(), and can use the function write.table() to do the reverse. As a minimum, it requires as arguments the object to be written and the name of the file it is written to

> write.table(descriptiveStats.df.withNA,”DescriptiveStats.txt”)

Here, the extension “.txt” is given to the file, but you can provide other (or none). The code above produces a text file which looks like

“Experiment” “Mean” “n” “sd”

“1” “control” 2.75 4 1.25830573921179

“2” “treatment” 6.5 4 2.38047614284762

(remember you can easily open text files in your R (Studio) editor). Overriding some of the defaults is a good idea. In particular, we can avoid the quotations marks by specifying quote=FALSE and omit the row numbers that mess up the alignment by specifying row.names=FALSE. We can also specify we want the fields to be separated by a semi-colon using sep. Lastly, there is no need to have this many decimals and the function format() can be used to restrict the number of digits

> write.table(format(descriptiveStats.df.withNA,digits=3), “DescriptiveStats.txt”, quote=F, row.names = F, sep=”;”)

which produces the text file

Experiment;mean;n;sd

control;2.75;4;1.26

treatment;6.50;4;2.38

When you copy-paste the above text into Word, you can convert it to a Table (Highlight the text and select *Insert*->*Table*->*Convert Text to Table*; select “Semicolons” under “Separate Text at”). The Table will then in its basic form looks like:

|  |  |  |  |
| --- | --- | --- | --- |
| Experiment | mean | n | sd |
| Control | 2.75 | 4 | 1.26 |
| Treatment | 6.50 | 4 | 2.38 |

This basic table can then be formatted further in your document.

The approach (chapter 10.1) to create a data.frame by “picking” from objects what you want to have, adding these one by one can of course be applied to collect also other statistics. Results from statistical analyses, for example. The creation of this kind of tables by coding in R often feels quite cumbersome, especially when you want to combine many different statistics. In my experience, however, it is common to realise a (hopefully small) mistake was made at some point in the workflow. If the construction of your table was coded, you can then quickly update your table, at least compared to manually extracting the statistics you wanted to report. As a consequence, time spent in coding may be paid back to you with interest!

Another important aspect to remember is that even if the “line-by-line” coding quickly looks like a massive block of code, it is usually constructed of many repeats of the same line of code with only small alteration. For example in the above, we first create a start (with the mean per subset), and after that repeat the same procedure 2 times where only the name of the function is changed, but the lines of code are otherwise identical. Clearly, copy-pasting and carefully updating the name of the function allows one to efficiently create this kind of repetitive blocks of code fairly rapidly.

### 10.4 Exercises part III

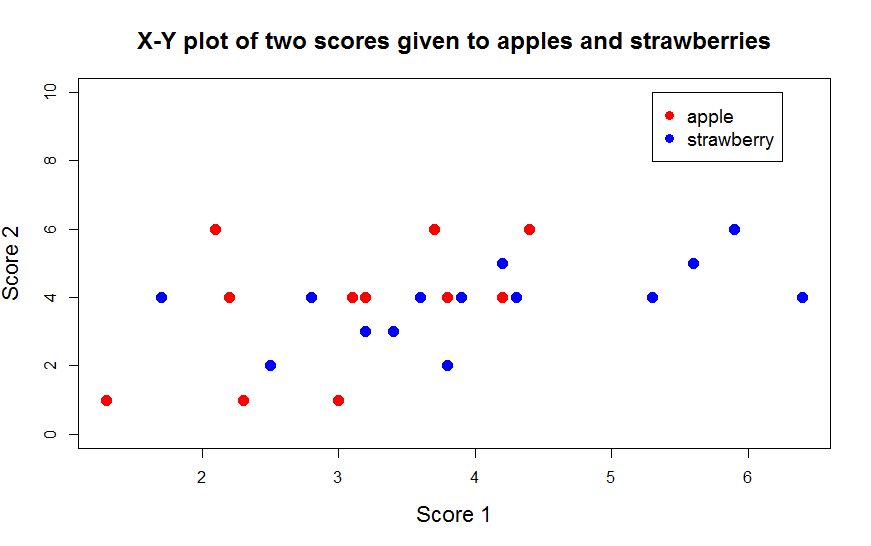
**Exercise III.1**

Use the data in “Fruit\_tasting\_data\_2018.txt” from Moodle

a) Pimp your plot, step by step (one line of code for each addition; remember that you can copy-paste your line of code and then add the next step to it)

* Plot “score 2” as a function of “score 1”
* Rename the axes
* Change the symbol plotted
* Increase point size of the symbol plotted
* Increase font size for the axes
* Change the y axis so that the minimum value is 0 and the maximum is 10
* Change the color of the points so that they are different for apples and for strawberries
* Add a text on top of the figure
* Add a legend

In the end, your figure should look something like this:



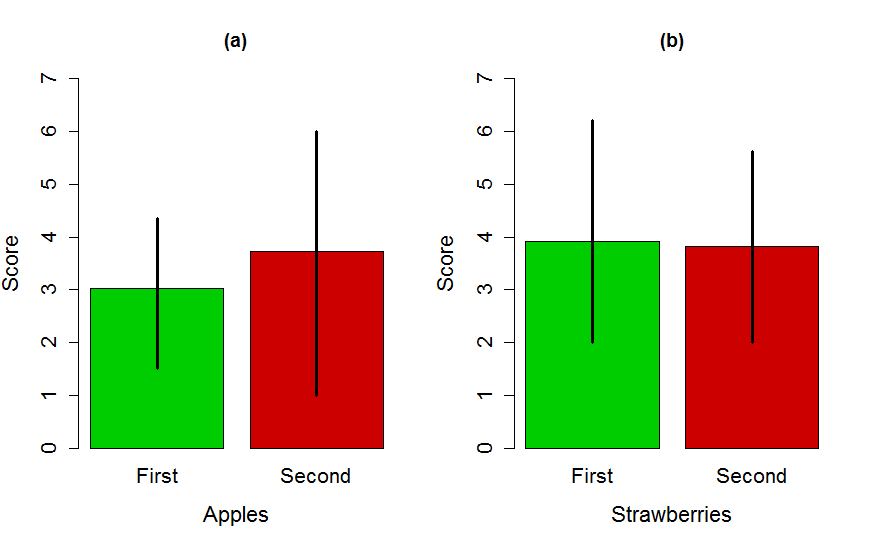
b. You obtain the extra scores 2.2 and 4 for the apple Gyllenkroks Astrakan. Add this to the data object in R. Redraw the plot

c. Code the exporting of the final plot as a PDF. Also export the plot as a PDF using the R Studio “Export” option. Are these two options producing the same outcome?

**Exercise III.2**

Use the data in “Fruit\_tasting\_data\_2018.txt” from Moodle

1. Calculate the mean and 95% quantile interval for score1 and score2 for apples and strawberries. The 95% quantile interval consists of 2 values. A lower and upper value. The 95% quantiles means that 95% of the data lies between these two values. The function quantile() is useful here (think carefully about which values to use as probability prob argument for this function).
2. Create a barplot of the mean score1 for apples and strawberries with a line indicating the 95% quantiles (i.e. the line goes from the lower value to the upper value, TIP: read carefully section 8.1 for this exercise). Make the plot look as good as you can.
3. Use par() to plot two panels in one graph and make a 2-panel plot showing the average score1 and score2 of apples and strawberries with their 95% quantile interval respectively. That is, one panel with a barplot of score1 and score2 of apples and the second panel showing barplot of score1 and score2 of strawberries. Something like this:



**Exercise** **III.3**

You learn from Wikipedia (<https://en.wikipedia.org/wiki/Standard_error>) that the standard error is defined as the standard deviation divided by the square root of the sample size.

1. Code a function called calc.SE() to calculate the standard error of any numerical vector passed to this function. By default, the function should not strip away NA elements in the vector (like in the function mean()), but it should also be possible to override the default and omit the NA values.

Check that it works: You should obtain the following output

> calc.SE(1:10)

[1] 0.9574271

> calc.SE(c(1:10,NA))

[1] NA

> calc.SE(c(1:10,NA),na.rm = TRUE)

[1] 0.9574271

1. The following lines make a small data.frame

df.year<-data.frame(Year=c(rep(1,5),rep(2,5)), Response=c(4.256898, 7.103097, 5.345175, 3.259264, 4.327387, 9.995211, 8.621862, 10.499859,10.032349, 8.821877))

Create this data.frame. Calculate the sample size, mean and SE of Response in both years in using both aggregate and tapply. Each of these calculations is ONE line of code (i.e. one line of code with aggregate to generate sample sizes per year, one to generate mean, etc..)

1. Some additional data is added which contains NA’s in your data.frame.

df.year<-rbind(df.year, data.frame(Year=c(rep(1,2),rep(2,2)), Response=c(4.33, NA, 10.1, NA)))

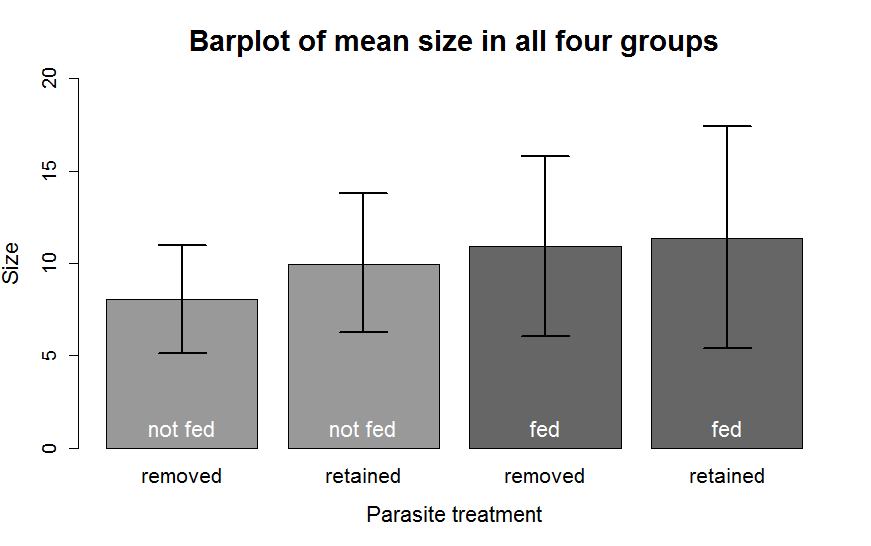
Run these lines to add this data, and calculate the sample size, mean and SE per year using both aggregate and tapply. Each of these calculations is ONE line of code (i.e. one line of code with aggregate to generate sample sizes per year, one to generate mean, etc..)

1. Combine the statistics you calculated in (c) above using either aggregate() or tapply() into a single data.frame that thus forms a table of all these descriptive statistics. Export the data.frame to make a nice looking table in a MS-WORD or Open office document. Something like:

|  |  |  |  |
| --- | --- | --- | --- |
| Year | n | mean | SE |
| 1 | 6 | 4.8 | 0.54 |
| 2 | 6 | 9.7 | 0.31 |

**Exercise III.4**

1. Extend your script of exercise II.2 to produce the following barplot of mean size +/- the 95% quantiles in each of the experimental groups in greyscale (You can find information on 95% quantiles in III.1). Make the graph look close to the graph below (note the way confidence intervals are plotted here; check R help or reference card for arrows()).



**Exercise III.5**

The following lines of code generate a small dataset

df.withNA<-data.frame(Trial=1:10, Experiment=c(rep(“control”,5),rep(“treatment”,5)),Measure=c(1,3,4,NA,3,4,5,NA,8,9))

df.withNA<- data.frame(df.withNA,Measure2=c(2,3,2,3,4,5,4,5,6,6),block=rep(c(“B1”,”B2”),5))

which when printed looks like:

Trial Experiment Measure Measure2 block

1 1 control 1 2 B1

2 2 control 3 3 B2

3 3 control 4 2 B1

4 4 control NA 3 B2

5 5 control 3 4 B1

6 6 treatment 4 5 B2

7 7 treatment 5 4 B1

8 8 treatment NA 5 B2

9 9 treatment 8 6 B1

10 10 treatment 9 6 B2

1. Use one line of code with aggregate()to compute how many different blocks there were per experiment group. The correct answer is not 5 per group.
2. Code the construction of a single data.frame that holds the descriptive statistics sample size, mean and standard deviation for “Measure” for each of the four possible combinations of “Experiment” and “block”. The code should add each descriptive statistic one at a time.
3. Code the construction of a single data.frame with the descriptive statistics sample size, mean and standard deviation for “Measure2” for each of the four possible combinations of “Experiment” and “block”. The code should add each descriptive statistic one at a time.
4. Code the construction of a single data.frame with the descriptive statistics sample size, mean and standard deviation for “Measure” and “Measure2”. The code can add each descriptive statistic one at a time, but has to use a single call to aggregate() to calculate each statistic for each of the four possible combinations of “Experiment” and “block”. Alternatively, you can code all in one line.
5. Are there differences between what you get in a,b,c,d? If so, why is this?