**Chapter 2 Basics**

**http://qpolr.com/basics.html#missing-values-na**

Remember that everything you do in R can be written as commands. Repeat what you did in the last chapter from your script window: write 2+2 and run the code (mark the code and press CTRL+R on Windows or CMD+ENTER on Mac). This should look like the output below.

2+2

## [1] 4

You are now able to conduct simple arithmetics. This shows that R can be used as a calculatur and you can now call yourself an R user (go put that on your CV before you continue). In other words, knowing how to use R is not a binary category where you either can use R or not, but a continuum where you will always be able to learn more. That’s great news! However, that also means that you will always be able to learn more.

**2.1 Numbers as data**

Next, we will have to learn about variable assignments and in particular how we can work with *objects*. Everything you will use in R is saved in objects. This can be everything from a number or a word to complex datasets. A key advantage of this, compared to other statistical programmes, is that you can have multiple datasets open at the same time. If you, for example, want to connect two different surveys, you can have them both loaded in the memory at the same time and work with them. This is not possible in SPSS or Stata.

To save something in an object (e.g. a variable), we need to use the *assignment operator*, <-, which basically tells R that anything on the right side of the operator should be assigned to the object on the left side. Let us try to save the number 2 in the object x.

x <- 2

Now x will return the number 2 whenever we write x. Let us try to use our object in different simple operations. Write the operations below in your R-script and run them individually and see what happens.

x

x \* 2 # x times 2

x \* x # x times x

x + x # x plus x

If it is working, R should return the values 2, 4, 4 and 4. If you change the object x to have the number 3 instead of 2 (x <- 3) and run the script again, you should get different results.[1](http://qpolr.com/basics.html#fn1) This is great as you only need to change a single number to change the output from the whole procedure. Accordingly, when you are working with scripts, try to save as much you can in objects, so you only need to change information once, if you want to make changes. This will reduce the likelihood of making mistakes.

We can also use our object to create other objects. In the example below, we will create a new object y. This object returns the sum of x and 7.

y <- x + 7

One thing to keep in mind is that we do not get the output in y right away. To get the output, we can type y.

y

## [1] 9

Alternatively, when we create the object, we can include it all in a parenthesis as we do below. This tells R that we do not only want to save some information in the object y, but that we also want to see what is saved in y.

(y <- x + 7)

## [1] 9

Luckily, we are not limited to save only one number in an object. On the contrary, in most objects we will be working with, we will have multiple numbers. The code below will return a row of numbers from 1 to 10.

1:10

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

We can save this row of numbers in an object (again using <-), but we can also work with them directly, e.g. by taking every number in the row and add 2 to all of them.

1:10 + 2

## [1] 3 4 5 6 7 8 9 10 11 12

When you will be working with more numbers, you have to tell R that you are working with multiple numbers. To do this, we use the function c(). This tells R that we are working with a vector.[2](http://qpolr.com/basics.html#fn2) The function c() is short for *concatenate* or *combine*. Remember that everything happening in R happens with functions. A vector can look like this:

c(2, 2, 2)

## [1] 2 2 2

This is a *numerical* vector. A vector is a collection of values of the same type.[3](http://qpolr.com/basics.html#fn3) We can save any vector in an object. In the code below we save four numbers (14, 6, 23, 2) in the object x.

# save 14, 6, 23 and 2 in the object x

x <- c(14, 6, 23, 2)

# show the output of x

x

## [1] 14 6 23 2

We can then use this vector to calculate new numbers (just as we did above with 1:10), for example by multiplying all the numbers in the vector with 2.

# calculate x times 2

x \* 2

## [1] 28 12 46 4

If we are only interested in a single value from the vector, we can get this value by using brackets, i.e. [ ], which you place just after the object (no space between the name of the object and the brackets!). By placing the number 3 in the brackets we get the third number in the object.

x[3]

## [1] 23

As you can see, we get the third element, 23. We can use the same procedure to get all values with the exception of one value by including a negative sign in the brackets. In the example below we will get all values except for 2. Also, note that since we are not assigning anything to an object (with <-), we are not making any changes to x.

x[-2]

## [1] 14 23 2

Now we can try to use a series of functions on our object. The functions below will return different types of information such as the median, the mean, the standard deviation etc.

length(x) # length of vector, number of values

min(x) # minima value

max(x) # maxima value

median(x) # the median

sum(x) # the sum

mean(x) # the mean

var(x) # the variance

sd(x) # the standard deviation

The functions should return the values 4, 2, 23, 10, 45, 11.25, 86.25 and 9.287088.

If we for some reason wants to add an extra number to our vector x, we can either create a new vector with all the numbers or just overwrite the existing vector with the addition of an extra number:

x <- c(x, 5)

x

## [1] 14 6 23 2 5

We now have five values in our vector instead of four. The value 5 has the last place in the vector but if we had added 5 before x in the code above, 5 would have been in the beginning of the vector.

Try to use the mean() function on the new object x

mean(x)

## [1] 10

Now the mean is 10 (before we added the value 5 to the object the mean was 11.25).

**2.2 Missing values (NA)**

Up until now we have been lucky that all of our “data” has been easy to work with. However, in the real world - and thereby for most of the data we will work with - we will encounter missing values. In Stata you will see that missing values get a dot (‘.’). In R, all missing values are denoted NA. Let us try to add a missing value to our object x and take the mean.

x <- c(x, NA)

mean(x)

## [1] NA

We do not get a mean now but just NA. The reason for this is that R is unable to calculate the mean of a vector with a missing value included. In order for R to calculate the mean now, we need to specify that it should remove the missing values before calculating the mean. To do this, we add na.rm=TRUE as an *option* to the function. Most functions have a series of options (more on this later), and the default option for the mean() function is not to ignore the missing values.

mean(x, na.rm=TRUE)

## [1] 10

Now we get the same mean as before we added NA to the object.

**2.3 Logical operators**

In R a lot of what we will be doing is using logical operators, e.g. testing whether something is equal or similar to something else. This is in particular relevant when we have to recode objects and only use specific values. If something is true, we get the value TRUE, and if something is false, we get FALSE. Try to run the code below and see what information you get (and whether it makes sense).

x <- 2

x == 2 # equal to

x == 3

x != 2 # not equal to

x < 1 # less than

x > 1 # greater than

x <= 2 # less than or equal to

x >= 2.01 # greater than or equal to

The script will return TRUE, FALSE, FALSE, FALSE, TRUE, TRUE and FALSE. If you change x to 3, the script will (logically) return other values.

**2.4 Text as data**

In addition to numbers we can and will also work with text. The difference between text and numbers in R is that we use quotation marks to indicate that something is text (and not an object).[4](http://qpolr.com/basics.html#fn4) As an example, we will create an object called p with the political parties from the United Kingdom general election in 2017.

p <- c("Conservative Party", "Labour Party", "Scottish National Party",

"Liberal Democrats", "Democratic Unionist Party", "Sinn Féin")

p

## [1] "Conservative Party" "Labour Party" "Scottish National Party" "Liberal Democrats"

## [5] "Democratic Unionist Party" "Sinn Féin"

To see what type of data we have in our object, p, we can use the function class(). This function returns information on the type of data we are having in the object. If we use the function on p, we can see that the object consists of characters (i.e. *“character”*).

class(p)

## [1] "character"

To compare, we can do the same thing with our object x, which includes numerical values. Here we see that the function class() for x returns "numeric". The different classes a vector can have are: character (text), numeric (numbers), integer (whole numbers), factor (categories) and logical (logical).

class(x)

## [1] "numeric"

To test whether our object is numerical or not, we can use the function is.numeric(). If the object is numeric, we will get a TRUE. If not, we will get a FALSE. This logical structure can be used in a lot of different scenarios (as we will see later). Similar to is.numeric(), we have a function called is.character() that will show us whether the object is a charater or not.

is.numeric(x)

is.character(x)

Try to use is.numeric() and is.character() on the object p.

To get the number of characters for each element in our object, we can use the function nchar():

nchar(p)

## [1] 18 12 23 17 25 9

We can also convert the characters in different ways. First, we can convert all characters to uppercase with toupper(). Second, we can concert all characters to lowercase with tolower().

toupper(p)

## [1] "CONSERVATIVE PARTY" "LABOUR PARTY" "SCOTTISH NATIONAL PARTY" "LIBERAL DEMOCRATS"

## [5] "DEMOCRATIC UNIONIST PARTY" "SINN FÉIN"

tolower(p)

## [1] "conservative party" "labour party" "scottish national party" "liberal democrats"

## [5] "democratic unionist party" "sinn féin"

In the same way we could get specific values from the object when it was numeric, we can get specific values when it is a character object as well.

p[3]

## [1] "Scottish National Party"

p[-3]

## [1] "Conservative Party" "Labour Party" "Liberal Democrats" "Democratic Unionist Party"

## [5] "Sinn Féin"

While p is a short name for an object and easy to write, it is not telling for what we actually have stored in the object. Let us create a new object called party with the same information as in p. When you name objects remember that they are case sensitive so party will be a different object than Party.[5](http://qpolr.com/basics.html#fn5)

party <- p

party

## [1] "Conservative Party" "Labour Party" "Scottish National Party" "Liberal Democrats"

## [5] "Democratic Unionist Party" "Sinn Féin"

**2.5 Data frames**

In most cases, we will not be working with one variable (e.g. information on party names) but multiple variables. To do this in an easy way, we can create *data frames* which is similar to a dataset in SPSS and Stata. The good thing about R, however, is that we can have multiple data frames open at the same time. The cost of this is that we have to specify, when we do something in R, exactly what data frame we are using.

Here we will create a data frame with more information about the parties from the United Kingdom general election, 2017.[6](http://qpolr.com/basics.html#fn6)

As a first step we can create new objects with more information: leader (information on the party leader), votes (the vote share in percent), seats (the number of seats) and seats\_change (change in seats from the previous election). Do note that the order is important as we are going to link these objects together in a minute, where the first value in each object is for the Conservative Party, the second for the Labour Party and so on.

party <- c("Conservative Party", "Labour Party", "Scottish National Party",

"Liberal Democrats", "Democratic Unionist Party", "Sinn Féin")

leader <- c("Theresa May", "Jeremy Corbyn", "Nicola Sturgeon",

"Tim Farron", "Arlene Foster", "Gerry Adams")

votes <- c(42.4, 40.0, 3.0, 7.4, 0.9, 0.7)

seats <- c(317, 262, 35, 12, 10, 7)

seats\_change <- c(-13, 30, -21, 4, 2, 3)

The next thing we have to do is to connect the objects into a single object, i.e. our data frame. A data frame is a collection of different vectors of the same length. In other words, for the objects we have above, as they have the same number of information, they can be connected in a data frame. R will return an error message if the vectors do not have the same length.

We can have different types of variables in a data frame, i.e. both numbers and text variables. To create our data frame, we will use the function data.frame() and save the data frame in the object uk2017.

uk2017 <- data.frame(party, leader, votes, seats, seats\_change)

uk2017 # show the content of the data frame

## party leader votes seats seats\_change

## 1 Conservative Party Theresa May 42.4 317 -13

## 2 Labour Party Jeremy Corbyn 40.0 262 30

## 3 Scottish National Party Nicola Sturgeon 3.0 35 -21

## 4 Liberal Democrats Tim Farron 7.4 12 4

## 5 Democratic Unionist Party Arlene Foster 0.9 10 2

## 6 Sinn Féin Gerry Adams 0.7 7 3

To see what type of object we are working with, we can use the function class() again to show that uk2017 is a data frame.

class(uk2017)

## [1] "data.frame"

If we would like to know what class the individual variables in our data frame are, we can use the function sapply(). This function allows us to apply a function to a list or a vector. Below we apply class() on the individual variables in uk2017.

sapply(uk2017, class)

## party leader votes seats seats\_change

## "factor" "factor" "numeric" "numeric" "numeric"

Here we can see that we have data as a factor as well as numerical variables. We can get similar information about our data by using the function str(). This function returns information on the structure of the data frame.

str(uk2017)

## 'data.frame': 6 obs. of 5 variables:

## $ party : Factor w/ 6 levels "Conservative Party",..: 1 3 5 4 2 6

## $ leader : Factor w/ 6 levels "Arlene Foster",..: 5 3 4 6 1 2

## $ votes : num 42.4 40 3 7.4 0.9 0.7

## $ seats : num 317 262 35 12 10 7

## $ seats\_change: num -13 30 -21 4 2 3

We can see that it is a data frame with 6 observations of 5 variables. If the rows (i.e. observations) have names, we can get these by using rownames(). We can get the names of the columns, i.e. the variables in our data frame, by using colnames().

colnames(uk2017)

## [1] "party" "leader" "votes" "seats" "seats\_change"

If we want to see the number of columns and rows in our data frame, we can use ncol() and nrow().

ncol(uk2017)

## [1] 5

nrow(uk2017)

## [1] 6

If we are working with bigger data frames, e.g. a survey with thousands of respondents, it might not be useful to show the full data frame. One way to see a few of the observations is by using head(). If not specified further, this function will show the first six observations in the data frame. In the example below, we will tell R to show the first three observations

head(uk2017, 3) # show the first three rows

## party leader votes seats seats\_change

## 1 Conservative Party Theresa May 42.4 317 -13

## 2 Labour Party Jeremy Corbyn 40.0 262 30

## 3 Scottish National Party Nicola Sturgeon 3.0 35 -21

In the same way, we can use tail() to show the last observations in a data frame. Here we see the last four observations in our data frame.

tail(uk2017, 4) # show the last four rows

## party leader votes seats seats\_change

## 3 Scottish National Party Nicola Sturgeon 3.0 35 -21

## 4 Liberal Democrats Tim Farron 7.4 12 4

## 5 Democratic Unionist Party Arlene Foster 0.9 10 2

## 6 Sinn Féin Gerry Adams 0.7 7 3

If you want to see your data frame in a new window, you can use the function View() (do note the capital letter V - not v). Again, R is very (case) sensitive.

View(uk2017)

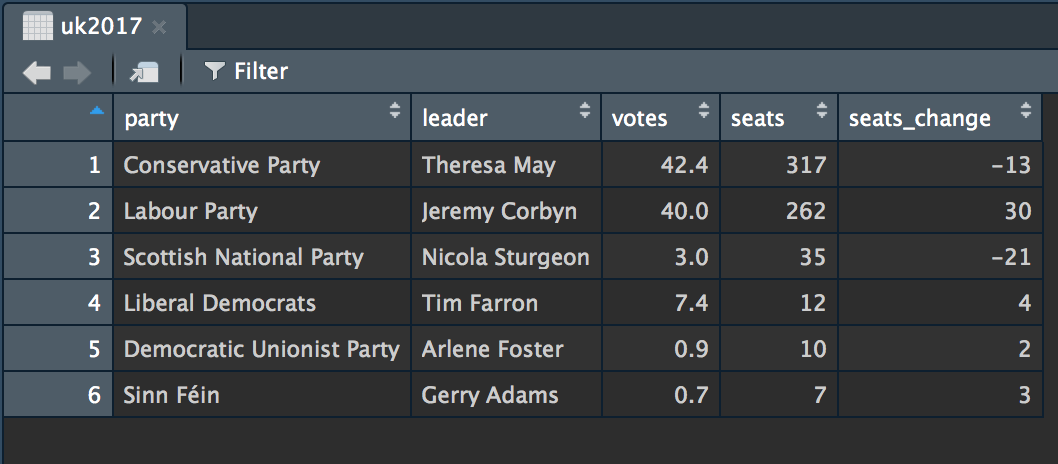


Figure 2.1: Data frame with View(), RStudio

When you are working with variables in a data frame, you can use $ as a *component selector* to select a variable in a data frame. This is the base R way, i.e. brackets and dollar signs. In the next chapter we will work with other functions that makes it easier to work with data frames.

If we, for example, want to have all the vote shares in our data frame uk2017, we can write uk2017$votes.

uk2017$votes

## [1] 42.4 40.0 3.0 7.4 0.9 0.7

Contrary to working with a vector in a single dimension, we have two dimensions in a data frame (rows horisontally and columns vertically). Just as for a single vector, we need to work with the brackets, [ ], in addition to our object. However, now we need to specify the rows *and* columns we are interested in. If we want to work with the first row, we need to specify [1, ] after the object. The comma is seperating the information on the rows and columns we want to work with. When we are not specifying anything after the comma, that means we want to have the information for *all* columns.

uk2017[1,] # first row

## party leader votes seats seats\_change

## 1 Conservative Party Theresa May 42.4 317 -13

Had we also added a number after the comma, we would get the information for that specific column. in the example below we want to have the information on the first row in the first column (i.e. the name of the party on the first row).

uk2017[1, 1] # first row, first column

## [1] Conservative Party

## 6 Levels: Conservative Party Democratic Unionist Party Labour Party Liberal Democrats ... Sinn Féin

If we want to have the names of all parties, i.e. the information in the first column, we can specify that we want all rows but only for the first column.

uk2017[, 1] # first column

## [1] Conservative Party Labour Party Scottish National Party Liberal Democrats

## [5] Democratic Unionist Party Sinn Féin

## 6 Levels: Conservative Party Democratic Unionist Party Labour Party Liberal Democrats ... Sinn Féin

Interestingly, the functions we have talked about so far can all be applied to data frames. The summary() function is very useful if you want to get an overview of all variables in your data frame. For the numerical variables in the data frame, the function will return information such as the mean and the median.

summary(uk2017)

## party leader votes seats seats\_change

## Conservative Party :1 Arlene Foster :1 Min. : 0.700 Min. : 7.0 Min. :-21.0000

## Democratic Unionist Party:1 Gerry Adams :1 1st Qu.: 1.425 1st Qu.: 10.5 1st Qu.: -9.2500

## Labour Party :1 Jeremy Corbyn :1 Median : 5.200 Median : 23.5 Median : 2.5000

## Liberal Democrats :1 Nicola Sturgeon:1 Mean :15.733 Mean :107.2 Mean : 0.8333

## Scottish National Party :1 Theresa May :1 3rd Qu.:31.850 3rd Qu.:205.2 3rd Qu.: 3.7500

## Sinn Féin :1 Tim Farron :1 Max. :42.400 Max. :317.0 Max. : 30.0000

We can also use the functions on our variables as we did above, e.g. to get the maximum number of votes a party got with the function max().

max(uk2017$votes)

## [1] 42.4

If we want to have the value of a specific variable in our data frame, we can use both $ and [ ]. Below we get the second value in the variable party.

uk2017$party[2]

## [1] Labour Party

## 6 Levels: Conservative Party Democratic Unionist Party Labour Party Liberal Democrats ... Sinn Féin

To illustrate how we can combine a lot of what we have used above, we can get informatin on the name of the party that got the most votes. In order to do this, we specify that we would like to have the name of the party for the party where the number of votes equals the maximum number of votes. In other words, when uk2017$votes is equal to max(uk2017$votes), we want to get the information on uk2017$party. We use the logical operator == to test whether something is equal to.

uk2017$party[uk2017$votes == max(uk2017$votes)]

## [1] Conservative Party

## 6 Levels: Conservative Party Democratic Unionist Party Labour Party Liberal Democrats ... Sinn Féin

As we can see, the Conservative Party got the most votes in the 2017 election. We can use the same procedure if we want to get information on the party that got the minimum number of votes. To do this we use min(). Here we can see that this is Sinn Féin in our data frame.

uk2017$party[uk2017$votes == min(uk2017$votes)]

## [1] Sinn Féin

## 6 Levels: Conservative Party Democratic Unionist Party Labour Party Liberal Democrats ... Sinn Féin

The sky is the limit when it comes to what we can do with data frames, including various types of statistical analyses. To give one example, we can use the lm() function to conduct an OLS regression with votes as the independent variable and seats as the dependent variable (more on this specific function in R later). First, we save the model in the object uk2017\_lm and then use summary() to get the results.

uk2017\_lm <- lm(seats ~ votes, data = uk2017)

summary(uk2017\_lm)

##

## Call:

## lm(formula = seats ~ votes, data = uk2017)

##

## Residuals:

## 1 2 3 4 5 6

## 20.890 -17.105 18.054 -36.122 7.933 6.350

##

## Coefficients:

## Estimate Std. Error t value Pr(>|t|)

## (Intercept) -4.310 13.405 -0.321 0.763932

## votes 7.085 0.558 12.698 0.000222 \*\*\*

## ---

## Signif. codes: 0 '\*\*\*' 0.001 '\*\*' 0.01 '\*' 0.05 '.' 0.1 ' ' 1

##

## Residual standard error: 24.81 on 4 degrees of freedom

## Multiple R-squared: 0.9758, Adjusted R-squared: 0.9697

## F-statistic: 161.2 on 1 and 4 DF, p-value: 0.0002216

The coefficient for votes is positive and statistically significant (p<0.05

). In other words, as the vote share increases, so does the number of seats.

Last, when working with some of the functions in later chapters, you will encounter that we do not talk about data frames but so-called *tibbles*. In brief, they are also data frames but with some improvements. For most of the procedures used in this book, however, it is not too important whether your object is a data frame or a tibble. We can convert our uk2017 data frame to a tibble with the as\_tible() function (available in the tidyverse package).

as\_tibble(uk2017)

## # A tibble: 6 x 5

## party leader votes seats seats\_change

## <fct> <fct> <dbl> <dbl> <dbl>

## 1 Conservative Party Theresa May 42.4 317 -13

## 2 Labour Party Jeremy Corbyn 40 262 30

## 3 Scottish National Party Nicola Sturgeon 3 35 -21

## 4 Liberal Democrats Tim Farron 7.4 12 4

## 5 Democratic Unionist Party Arlene Foster 0.9 10 2

## 6 Sinn Féin Gerry Adams 0.7 7 3

**2.6 Import and export data frames**

Most of the data frames we will be working with in R are not data frames we will build from scratch but on the contrary data frames we will import from other files such as files made for Stata, SPSS or Excel. The most useful filetype to use when you work with data in files is .csv, which stands for *comma-separated values*. This is an open file format and can be opened in any software. To export and import data frames to .csv files, we can use write.csv() and read.csv().

First of all we need to know where R is working from, i.e. what our *working directory* is. In other words, we need to tell R where it should be saving the file and - when we want to import a data frame - where to look for a file. To see where R is currently working from (the *working directory*) you can type getwd(). This will return the place where R is currently going to save the file if we do not change it.

getwd()

If you would like to change this, you can use the function setwd(). This function allows you to change the working directory to whatever folder on your computer you would like to use. In the code below I change the working directory to the folder book in the folder qpolr in the Dropbox folder. Do also note that we are using forward slash (/) and not backslash (\).

setwd("/Dropbox/qpolr/book")

If you cannot remember the destination, you can use the menu to find the folder you want to have as your working directory as shown in Figure [2.2](http://qpolr.com/basics.html#fig:setwd). On a Mac, you can also use Option + CMD + C to copy the pathway to a file.

Figure 2.2: How to change the working directory

An easy way to control the working directory is to open an R-script directly from the folder you want to have as your working directory. Specifically, instead of opening RStudio and finding the script, find the script in your folder and open RStudio that way. This will automatically set the working directory to the folder with the R-script.

Once we know where we will save our data, we can use write.csv() to save the data. In the code below we first specify that we want to save the data frame uk2017 and next the filename of the file (uk2017.csv).

write.csv(uk2017, "uk2017.csv")

Do note that we need to put the file in quotation marks. Next, we can import the file into R the next time we open R with the function read.csv() and save the data frame in the object uk2017.

uk2017 <- read.csv("uk2017.csv")

As with most stuff in R, there are multiple ways of doing things. To import and export data, we have packages like foreign (R Core Team, 2015), rio (C. Chan, Chan, & Leeper, 2016) and readr (H. Wickham & Francois, 2015). If you install and load the package rio, you can use the functions import() and export().

# export data with the rio package

export(uk2017, "uk2017.csv")

# import data with the rio package

uk2017 <- import("uk2017.csv")

**2.7 Environment**

We have worked with a series of different objects. To see what objects we have in our memory, we can look in the *Environment* window, but we can also use the function ls()(*ls* is short for *list objects*).

ls()

## [1] "leader" "p" "party" "seats" "seats\_change" "uk2017" "uk2017\_lm"

## [8] "votes" "x" "y"

If we would like to remove an object from the memory, we can use the function rm() (*rm* is short for *remove*). Below we use rm() to remove the object x and then ls() to check whether x is gone.

rm(x)

ls()

## [1] "leader" "p" "party" "seats" "seats\_change" "uk2017" "uk2017\_lm"

## [8] "votes" "y"

If you would like to remove *everything* in the memory, you can use ls() in combination with rm().

rm(list = ls())

ls()

1. More specifically, 3, 6, 9 and 6.[↩](http://qpolr.com/basics.html#fnref1)
2. In the example with 1:10, this is similar to writing c(1, 2, 3, 4, 5, 6, 7, 8, 9, 10) and c(1:10). In other words, we have a hidden c() when we type 1:10.[↩](http://qpolr.com/basics.html#fnref2)
3. c() creates a vector with *all* elements in the parenthesis. Since a vector can only have one type of data, and not both numbers and text (cf. the next section), c() will ensure that all values are reduced to the level all values can work with. Consequently, if just one value is a letter and not a number, all values in the vector will be considered text.[↩](http://qpolr.com/basics.html#fnref3)
4. Alternatively, you can use ’ instead of “. If you want more information on when you should use ’ instead of”, see <http://style.tidyverse.org/syntax.html#quotes>.[↩](http://qpolr.com/basics.html#fnref4)
5. If you want more information on how to name objects, see <http://style.tidyverse.org/syntax.html#object-names>.[↩](http://qpolr.com/basics.html#fnref5)
6. The information is taken from <https://en.wikipedia.org/wiki/United_Kingdom_general_election,_2017>[↩](http://qpolr.com/basics.html#fnref6)