Data Handling in R: Exercise 1

Version 4: April 2014

Authored by Dr. Duncan Smallman, on behalf of EDINA and Data Library, University of Edinburgh as part of the Research Data MANTRA (Management Training) Project, funded by JISC MRD Programme (2010-11; http://www.ed.ac.uk/is/data-library-projects/mantra) and revised by Laine Ruus and Pauline Ward April 2014.

Contents

Data sets

Initially you will be creating your own objects (as guided in the practical) and will then use a spreadsheet to create a .txt and .csv file using the following river water quality data set from DEFRA:

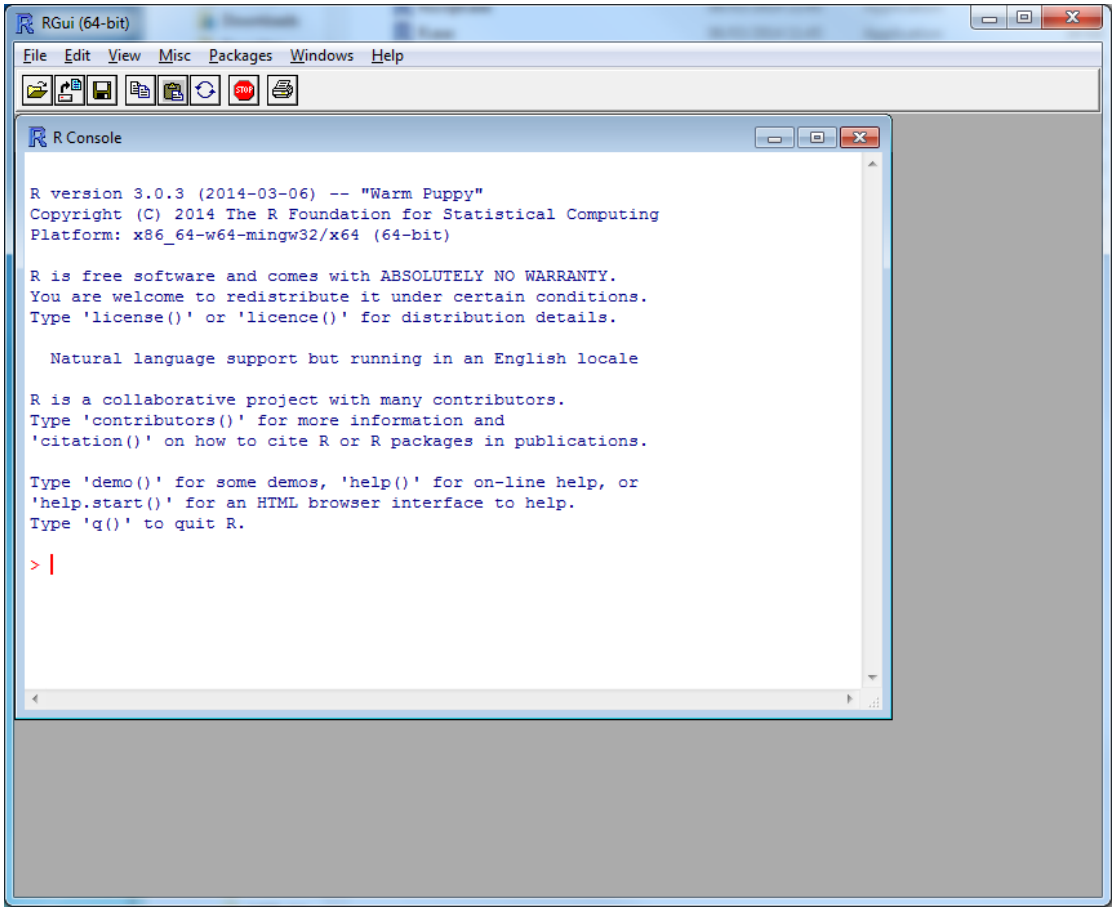
> "Department for Environment, Food and Rural Affairs (DEFRA) (2012): Annual average concentrations of selected determinands of river water quality, by river location: 1980 to 2011" at: http://data.gov.uk/dataset/annual-average-selected-determinands-of-river-water-quality-by-river-location
> (filename: iwtb10-deter-bylocation-201211.csv).

The R interface

This exercise is all about getting a feel for R and some of the basic functions for inputting data into R. By the end of this exercise you should be comfortable with writing syntax in R and inputting data into R. For a more detailed introduction to R itself, you may wish to refer to the manual which accompanies the software download (see file:///C:/Program%20Files/R/R-3.0.3/doc/manual/R-intro.html).

The first step is setting up R in a manner that is easy for you to use. Download and install the software via www.r-project.org. Initially when you open R you will be presented with a screen as shown in figure 1.1.

Figure 1.1 The RGui (i.e. interface) when first opened.

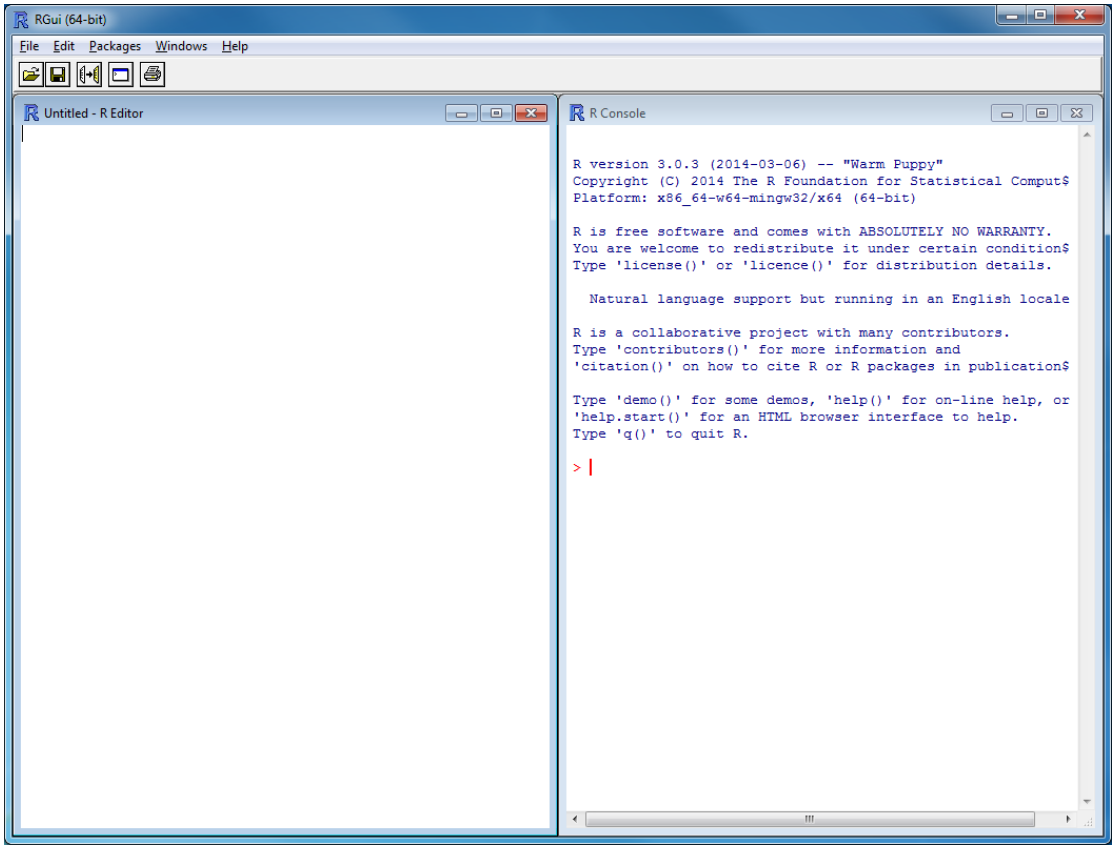


The R console is where you input the code for what you want to do and where all the output is printed.

Entering code repeatedly in R is quite laborious and also means you cannot easily keep track of what you have done. There is no one right way around this problem. One way is to write all the code you wish to use in a word processing package and then copy and paste it into R. Another to use the script editor already available in R as it allows you to run the line of code written straight into R without copy and pasting.

To open the script editor, click on "File" followed by "New Script". Initially the result may not look very practical but by selecting "Tile Vertically" under the "Windows" menu you end up with a view as shown in figure 1.2. This makes it easier to see what you have written along with the console.

Figure 1.2: A possible configuration of the Script Editor and R Console.



The Script written in the Editor can be saved as a text file and is the best place to record what you have done along with any output.

Behind the Script Editor and the R Console is your current workspace, containing objects created in the current session. An image of the workspace can also be stored at the end of an R session, to be reloaded the next time you start R.

Ways of inputting data

Let's start with a simple input of data. R can act as a calculator but generally any objects in R will need a name attached. To assign anything as an object we use "<-". In its simplest form an object containing a sequence of numbers can be assigned thus by entering:

```
numberlist<-1:10
numberlist
```

into the script editor and click the "Run line or selection" button on the right of the "Save" short cut key after each line. 

An equally fast method which allows a greater degree of specification is to use "`seq()`" and specify the start number, the last number and the intervals:

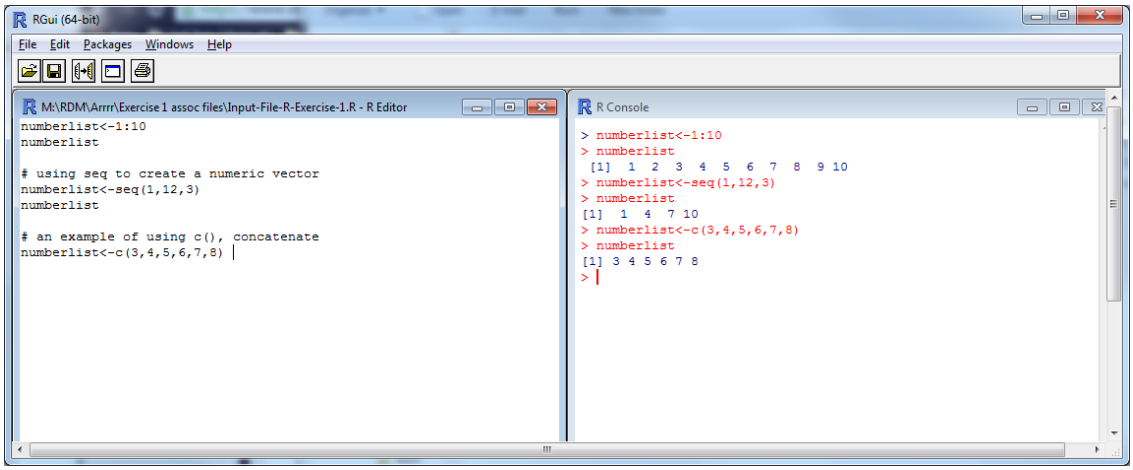
```
#using seq to create a numeric vector
numberlist<-seq(1,12,3)
numberlist
```

An alternative (and slightly long winded) method is to use concatenate <– `c()` as below (the "c" means concatenate). The concatenate function, c(), is one of the most used and useful tools. In a way it creates an object that can be used in larger functions. It is a way of entering text to create factors, labels or objects for equations or specifying dimensions. We will be using c() frequently throughout these exercises. Generally if there is a problem because some label is not recognised, then c() can be used, as we shall see in later sections. In the script window enter:

```
#an example of using c(), concatenate
numberlist<-c(3,4,5,6,7,8)
numberlist
```

And click on the "Run Selection or Line" button after each entry. The result should look like figure 1.3.

Figure 1.3: Using concatenate, c(), to create a vector.



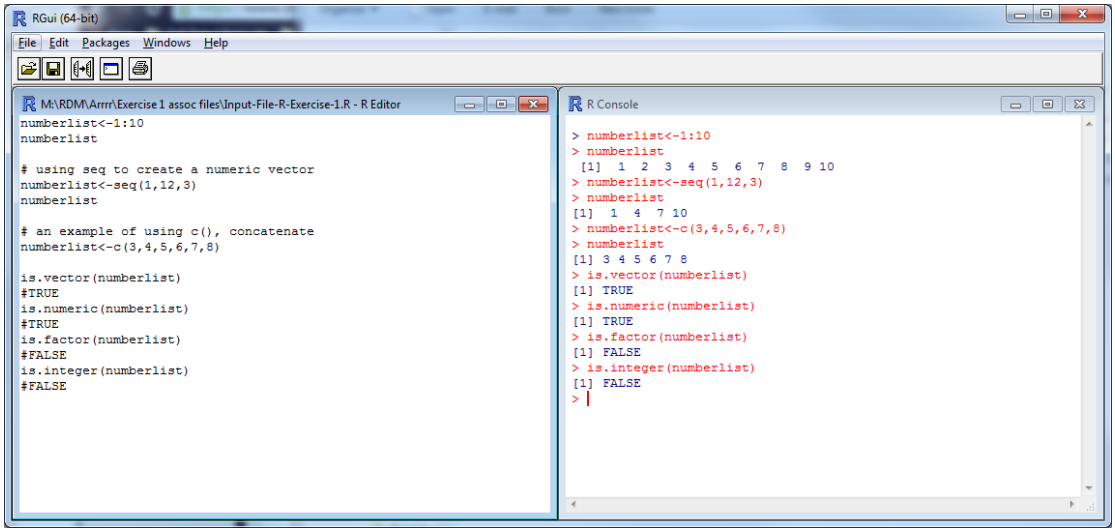
In all three cases we have created a numeric vector. There are ways of determining the properties of vectors such as whether it is a factor (a vector of labels which are words or alphanumeric codes) or a numeric vector (a vector of numbers such as numerical data); R treats factors (i.e. nominal or ordinal variables) and numeric vectors differently in analyses. To determine the type of a vector in R, we use "`is.`". For example we can enter in the script editor:

```
is.vector(numberlist)
is.numeric(numberlist)
is.factor(numberlist)
is.integer(numberlist)
```

By selecting all the entries we can send the selection all at once and get the result as shown in figure 1.4.

It is good practice to include the output within the script (by copying and pasting from the R console window) after giving a command. So we know that numberlist is a numeric vector (not a vector of factors) and that R considers the intervals to be on a continuous scale.

Figure 1.4: Selecting and sending multiple lines to the R Console



It is possible to change the properties of vectors by using "`as.`", for example:

```
#introduction to changing properties of objects
numberlist<-as.integer(numberlist)
```

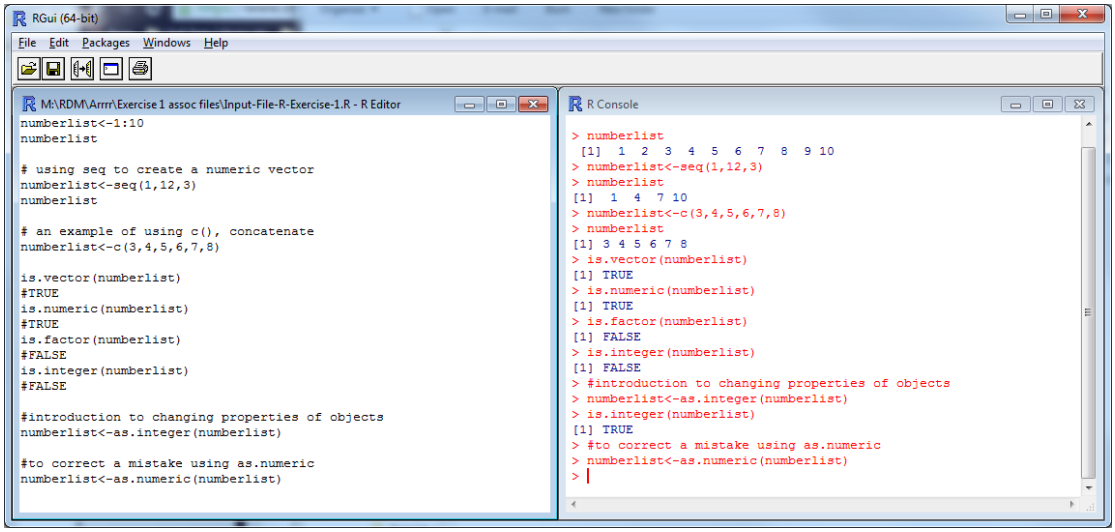
Now when we run `is.integer(numberlist)` again (use the up arrow on your keyboard to access recently-used commands from the R Console) we find that numberlist is an integer.

From this we can tell that any objects created in R are easily overwritten and that no undo function exists. This is IMPORTANT and demonstrates why keeping a good record of all commands used is important. Suppose we made a mistake in changing numberlist to a vector of integers. It is easily rectified by using:

```
# to correct a mistake using as.numeric
numberlist<-as.numeric(numberlist)
```

The screen shot in figure 1.5 shows the above steps and how useful it is to keep notes in between commands. The # indicates a comment and means if the text is sent to the console it is not read as code.

Figure 1.5: Using the script editor to keep notes and save command entries.



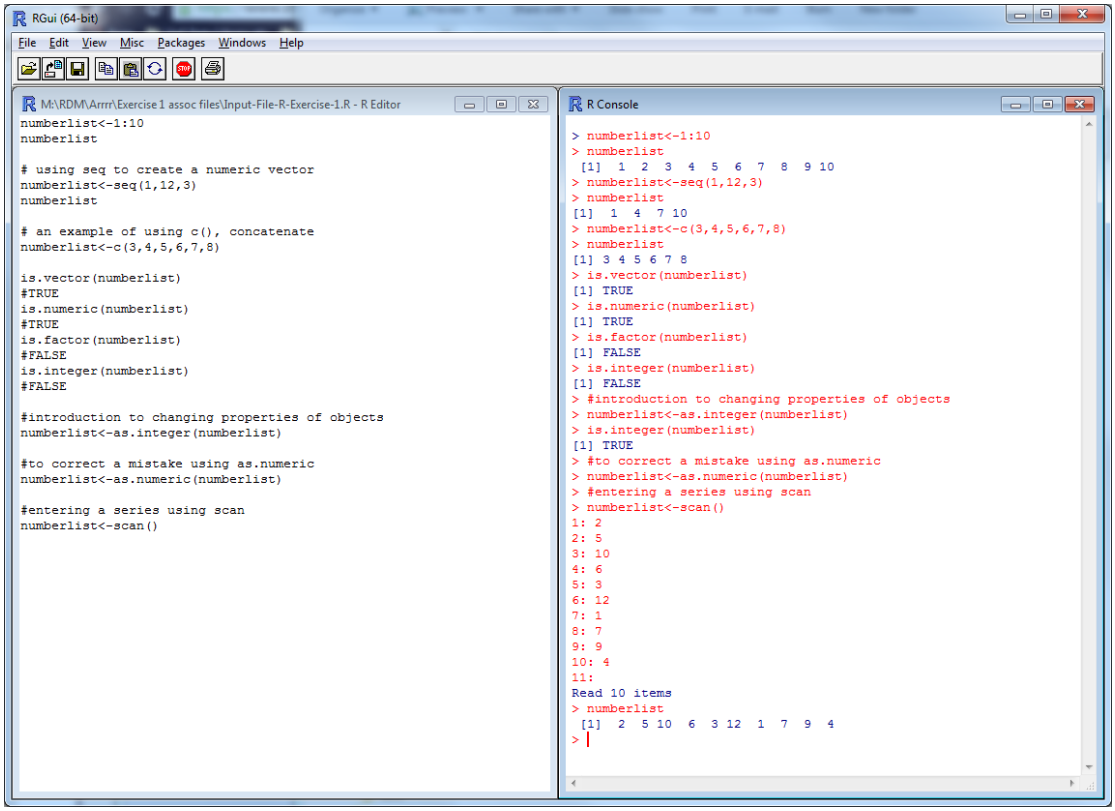
An alternative way to input a long list of numbers is using `scan()`. We'll create the vector numberlist again:

```
#entering a series of numbers for practice purposes
numberlist<-scan()
```

Now in the R Console window, enter any number and press enter after each number. After ten numbers, press enter twice. This will inform R that the entry has finished.

The result should look like the screen as shown in figure 1.6. Using scan is an easier way of entering data into R that allows you to keep track of what you entered. But this would be quite laborious if you were entering the corresponding factors (such as if the numbers related to specific location names or species).

Figure 1.6: Screen shot of using scan to create a numeric vector



Using "`rep`" allows you to specify what to repeat and how many times something should be repeated. The order the instructions are entered are: *what to repeat, how many of each, how many repeats in total*. For example:

Numberlist may relate to scores for an overall measure of quality of two rivers. The first five numbers relate to the Tyne and the next five relate to the Tweed.

```
#using rep () to create a factor
factorlist<-as.factor(rep(c("Tyne","Tweed"),each=5,times=1))
```

Then enter:

```
levels(factorlist)
```

This will inform you that you have just the two, Tyne and Tweed. By entering:

```
factorlist
```

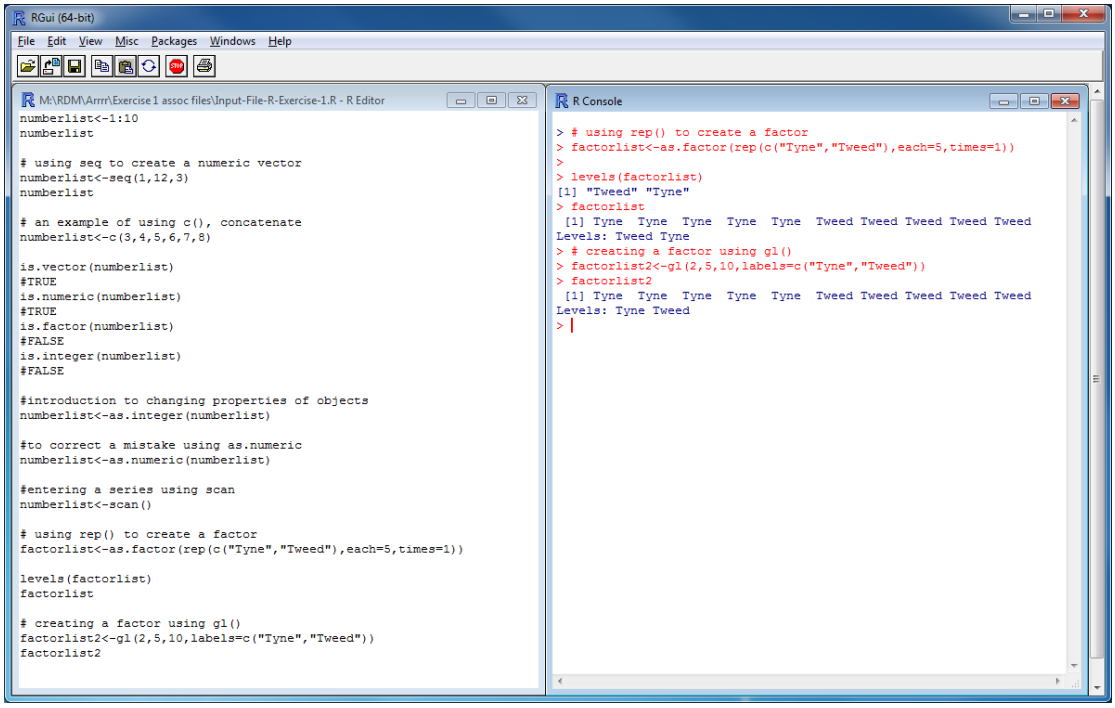
you will see the order in which the factors sit.

Another method to generate a list of factors is to use the generate factor levels function, `gl()`:

```
# creating a factor using gl()
factorlist2<-gl(2,5,10,labels=c("Tyne","Tweed"))
factorlist2
```

An example of using rep and gl to create factor vectors is shown in figure 1.7.

Figure 1.7: Creating a factor using rep and gl



Using `rep` can seem unwieldy but it does give you a degree of flexibility if the number of data points varies between factor levels or if you are using it to create a numeric dependent variable. Two examples should help explain this.

1) The vector numberlist refers to four rivers: Tyne, Tweed, Wear and Tees. We have two points of data from each of the Tyne and Wear, and three from the Tweed and Tees. We can create a factor called riverlist to take this into consideration:

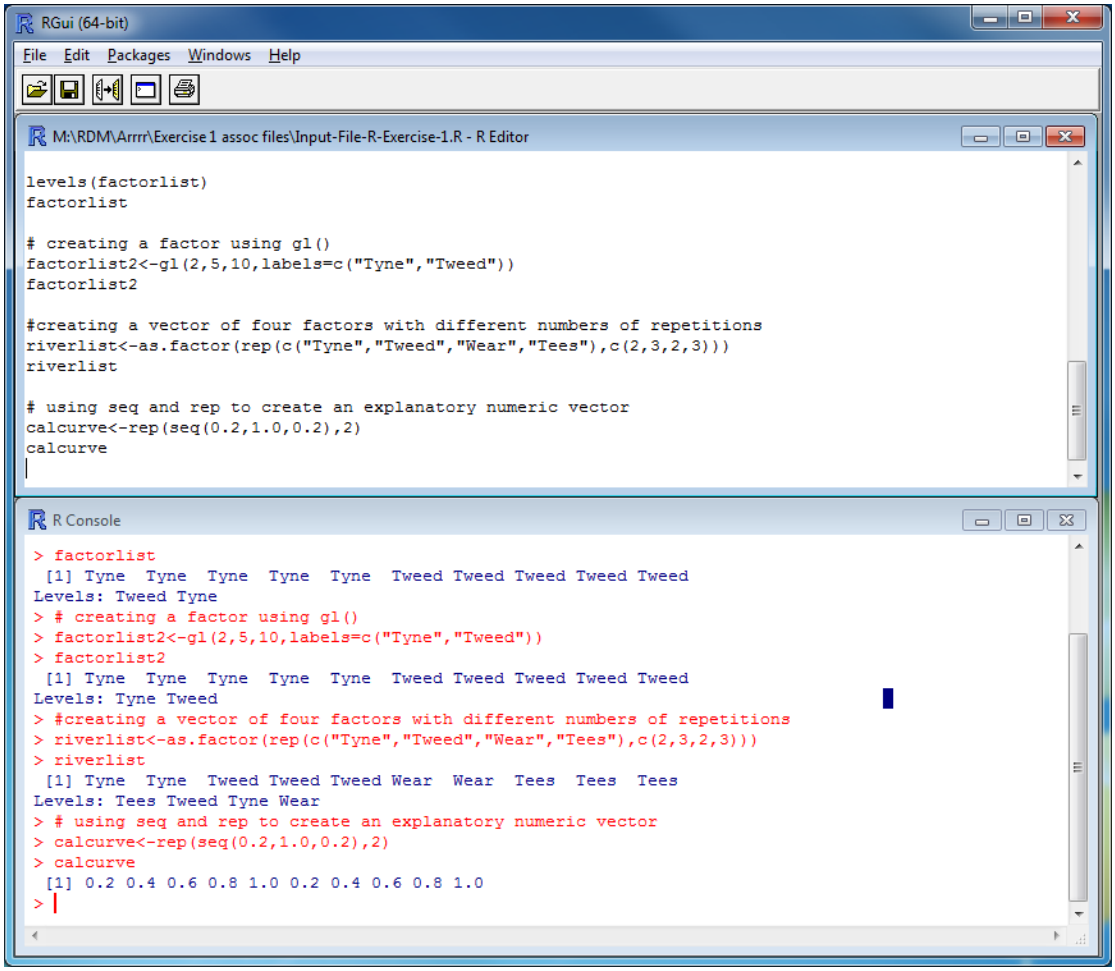
```
#creating a vector of four factors,
#with different numbers of repetitions
riverlist<-
as.factor(rep(c("Tyne","Tweed","Wear","Tees"),c(2,3,2,3)))
riverlist
```

2) We need to create an explanatory vector of known concentrations (i.e. for a calibration curve). Combined with the sequence function, `seq()`, `rep` becomes very useful:

```
# using seq and rep to create an explanatory numeric vector
calcurve<-rep(seq(0.2,1.0,0.2),2)
calcurve
```

The result should look like the screenshot in figure 1.8.

Figure 1.8: Creating a numeric vector using rep and seq



At this point it may be worth saving the script and the workspace. This is done by selecting the appropriate window and pressing the save button. If you close R, a prompt will appear asking if you wish to save the workspace image. If not previously saved selecting "Yes" is a sensible option. When R is opened again it will revert to the most recently saved workspace.

If carrying out different analyses on different sets of data or working across different data sets you may wish to save a workspace for each (in different subdirectories). If the last workspace you worked with is not the one you wish to work with, and you have saved a different workspace, you will need to load the workspace, using "load workspace" under the "File" tab.

When opening R, you will also need to open your previously saved script. To do this just go to "Open Script" under the "File" tab or the "Open script" short-cut tab in the window toolbar.

Using data frames and importing spreadsheet data

So far we have created separate objects that could be used for analysis purposes. The easiest way to handle several related data objects in R is using data frames. This is best seen by creating one. First let us look at how many active objects there are using the list function, `ls()`. `Ls()` will print the names of all the currently active objects within the workspace. To find out the number of active objects simply use `length()` in combination with `ls()`:

```
#using length and ls
#to determine the total number of active objects
length(ls())
```

```
[1] 5
```

The answer in this instance should be five, unless more objects have been created. We also need to confirm the names of the objects, by entering:

```
objects()
```

So now let's combine them into one data object:

```
#creating a data frame
exampleframe<-data.frame(factorlist,calcurve,numberlist)
names(exampleframe)
```

What we now have are the three objects created all held together in one object. It is always good to tidy up and keep the number of active objects to a minimum (hence why the script editor is useful, it is very easy to recreate objects as you

need them). We'll remove factorlist, calcurve and numberlist from our workspace using the remove function, `rm()`:

```
#removing factorlist, calcurve and numberlist from work space
rm(factorlist,calcurve,numberlist)
```

Now our data is held in one location, which makes it easier when coming to analyse it. Using `rm(ls())` removes all objects in the active workspace. This can be helpful if you start experiencing problems with "masking" or objects for some reason affect the outcome of analysis.

The easiest way to create a data frame is to import it from a spreadsheet programme such as MS Excel. The simplest format for importing the data is as a tab delimited text file (.txt suffix) or a comma separated file (comma delimited, .csv suffix). Certain rules apply when saving your data. If we enter "exampleframe" into R we get the outcome:

```
exampleframe
```

```
   factorlist calcurve numberlist
1        Tyne      0.2          2
2        Tyne      0.4          5
3        Tyne      0.6         10
4        Tyne      0.8          6
5        Tyne      1.0          3
6        Tees      0.2         12
7        Tees      0.4          1
8        Tees      0.6          7
9        Tees      0.8          9
10       Tees      1.0          4
```

This is how the data should be arranged in the spreadsheet before importing into R. The example we will use is a data set from DEFRA concerning

the annual average concentrations of selected determinants of river water quality, by location for the years 1980-2011. Download the file iwtb10-deter-bylocation-201211.csv. Open it using whatever spreadsheet software you prefer to work with (assuming in this instance MS Excel, you will need to select "All Files" from the file type drop box to locate the file).

For the import to be successful, please ensure:

- your data is in columns with the appropriate labels (factors) in columnar form;
- your variable names do not include spaces, (,), % or similar special characters (separators for words should be a ".");
- there are no rows containing text above or below your data table (you will need to delete several rows of text which appear before the column headings, and several which appear after the last row of comma-separated values in iwtb10-deter-bylocation-201211.csv).

N.B. Be aware that R is case sensitive. If there are blanks in your data, where data were not available, you will need to fill these in with NA if importing using `read.table()`. We will come to how R handles missing values shortly. If you are using MS Excel as your main spreadsheet software, and you are unfamiliar with how to save the data as a .csv or .txt file please read Box 1 . Make a careful note of where the files are saved as this is important when reading data into R.

Box 1: Saving an MS Excel spreadsheet as a .txt or .csv file.

Ensure that the criteria outlined in the main text for column names, extra rows etc have been met. Go to "File" and select "Save As". Select the most appropriate location to save the file in taking note of how many folders within the main drive (e.g. C:) the file sits. Click on the drop down menu called "Save as type" and select either csv (comma delimited) or txt (tab delimited). For this part of the exercise you will need to save a copy of the data as a csv and a tab delimited file. These will be referred to as RiverQuality.csv and RiverQuality.txt. A warning appears informing us that the file type does not support multiple worksheets. Click "OK". An information box now appears informing us that the file type may contain features not compatible with the file type. Click "Yes". And now the data is ready to be imported into R.

It is a personal preference which file format you use generally - .csv files are preferred for reasons that will become clear but either file type is fine.

We'll start with importing the RiverQuality.txt file using `read.csv()`. There are three key points to consider when reading data into R:

- Double quotes have to enclose the whole file path and file name.
- N.B. all instances of "\" in the file path must be replaced with "\\" e.g. "`M:\\RDM\\Arrrr\\RiverQuality.txt`".

This is where the script editor really does come in useful for noting which data you are importing and for having the file path ready for future use if you don't save the workspace. So let's do the following in the script editor, entering the file path relevant to where you saved the RiverQuality.txt file (the file path

used below is only an example and should not be copied directly). Read in the data using `read.csv()`:

```
riverquality<-
read.csv("M:\\RDM\\Arrrr\\RiverQuality.csv")
names(riverquality)
```

```
[1] "Region"   "River"    "Year"     "Temp"
 [5] "pH"       "Cond"     "SS"       "Ash"
 [9] "DO"       "BOD"      "Amm"      "Nitrite"
[13] "Nitrate"  "Chloride" "Alkaline" "Chloroph"
     [17] "Orthop"   "AnDet"
```

And now see the levels in Region:

```
levels(riverquality$Region)
```

Note that read.table() may be used for tab-delimited data. In this case, remember that any form of space or tab will be considered as a separate field in unless the separator (sep="\t") is explicitly stated, and the first row containing the variable names needs to be identified using header=T.

Missing values

Missing values within a data set are sometimes inevitable for various reasons. R has various ways of handling these. Missing values should be filled in as NA either directly into R or in the spreadsheet prior to reading into R. Most functions in R have a default setting of excluding NAs when carrying out analysis. There are two functions that can be used to either omit NA values (`na.omit`) or exclude them (`na.exclude`). The main difference is whether the NA values will still be taken into consideration as observations or removed completely during analysis.

NA should not be confused with NaN (Not a Number), which you are also likely to encounter though not actually within your data set. NaNs only occur when the result of the calculation produces a non-numerical result. The best example is:

```
Inf/Inf
[1] NaN
```

Throughout the practical we will look at how to deal with NAs. It is possible to get a summary of which of your variables in the data frame have NAs and how many. Using the function `apply()` allows you to get a quick summary like this:

```
apply(apply(riverquality,2,is.na),2,sum)
```

```
 Region     River      Year      Temp        pH      Cond
      0         0         0         8         1        11
     SS       Ash        DO       BOD       Amm  Nitrite
      0       384        19        19         0        53
Nitrate  Chloride Alkaline Chloroph   Orthop    AnDet
      5        23       155       321         5       266
```

The '2's within the `apply` functions refer to selecting the columns (with 1 referring to the rows). The output indicates that for many of the variables no data was available. We will handle this problem as we come to it throughout the other exercises.

There are other objects besides vectors and pre-existing data files that can be converted to data frames or used directly for analysis. Two of these are matrices and arrays. A matrix is a numeric object with 2 dimensions:

```
matexample<-matrix(1:9,nrow=3)
matexample
```

```
>     [,1] [,2] [,3]
```

```
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
```

The difference between a matrix and a data frame is that in a matrix all columns must have the same mode (factor, numeric, integer, etc) and must have the same length, as must the rows. A data frame can contain columns of different modes.

An array is a numeric object usually with 3 or more dimensions. We set the dimensions using `dim()`. Here we will generate an array with three dimensions with 24 random integers from a Poisson distribution (using rpois, random numbers from a poisson distribution) with mean=2.

```
Arrayexample<-rpois(24,2)
dim(Arrayexample)<-c(4,2,3)
Arrayexample
```

The result should be similar to the screen shot in figure 1.11.

Both objects are alternate methods of entering and storing data within R. It is important to note that the numbers are entered column-wise.

Figure 1.9: An example of creating an array based on 24 random integers from a Poisson distribution with mean=2