Data handling in R: Exercise 2

Version 4: April 2014
Authored by Dr Duncan Smallman, Dr Laine Ruus and Pauline Ward 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

Contents
Data handling in R: Exercise 2 .............................................................................................................. 1
Contents .................................................................................................................................................. 1
Introduction ........................................................................................................................................... 3
Exploring your data with plots .............................................................................................................. 3
  Figure 2.1: Value plot of the variable Temp from the riverquality data frame .... 4
  Figure 2.2: Value plot of the variable AnDet from the riverquality data frame .... 6
  Figure 2.3: Pairs plot of the first 8 variables from the river quality data frame .... 9
Working with arrays and matrices ...................................................................................................... 11
  Figure 2.4: Labelling the different dimensions within an array using Arrayexample.......................... 15
  Figure 2.5: Adding columns and rows to an array. Example used is riverarray.. 18
This work is licensed under the Creative Commons Attribution 2.5 UK: Scotland License. To view a copy of this license, visit

http://creativecommons.org/licenses/by/2.5/scotland/
**Introduction**

In Exercise 1 we covered some of the basics of entering data into R, importing data from external .csv or tab-delimited files and generally got a feel for working in the R environment.

In this exercise we will cover how to check that data was entered correctly, how to correct errors with inputted data and how to add new data to existing data frames. We will be using the River Quality data set (RiverQuality.txt or RiverQuality.csv) imported in Exercise 1 and the example matrix and array we previously created.

**Exploring your data with plots**

Before analysing any data, it is best practice to explore the data fully using plots of the data. This is a very good way of checking for any apparent anomalous data entries. R is a powerful graphics package as it was originally designed for graphics. It is beyond the scope of this practical to focus on the graphic functions of R. The very basic function is `plot()` and the argument can be structured in two ways:

- `plot(x, y)` – a co-ordinate layout
- `plot(y~x)` – a “y as predicted by x” which is the same as the “response variable as predicted by an independent variable”.

In this instance each response variable is plotted as it appears in the data frame and can be entered as `plot(y)` or `plot(y~1)`. As an example we will use the River Quality data set.

For multiple actions on a data frame, `attach()` may be used to save having to enter the name of the data frame before each variable. However, be warned that only a copy of the variable will be attached to the search path, and it may cause confusion by masking the values in the original data frame. New values assigned to the actual data frame will not be visible until it is detached and attached again. Alternatively you can use `with()` to specify the data frame (try typing “help(with)” for more details). We’ll start with the variable Temp. If
you’ve closed the R GUI since completing Exercise 1, remember to load the workspace from the R console, so that the variable riverquality will be available.

```r
# introduction to plotting using with()
with(riverquality, plot(Temp))
```

Figure 2.1: Value plot of the variable Temp from the riverquality data frame

A new window opens with a graph showing all the temperature values. The rank or position of each value within the variable is shown along the x-axis.

Before we go on to discuss what the plot shows, there are a few useful functions relevant for using plots to have a quick look at your data. By clicking on the graphics window, new menu options will appear. Under the History menu selecting “Recording” will mean you can scroll through multiple graphs once they have been plotted.
If we consider the plot for temperature, most points are clearly visible with no very distinct outliers (anomalous data points). Towards the left hand side of the graph there is one point which is below 3°C. As it is only a few degrees lower than the next lowest value (towards the right hand side) of the graph, we would assume that we had not made a data entry mistake. To check which value is the apparent outlier we can use either `which()` or subscripts (using square brackets “[]”).

```r
# different methods for checking apparent outliers
with(riverquality, which(Temp<3))
which(riverquality$Temp<3)
riverquality$Temp<3
```

We get the same answer, given as the row number, 150, but shown differently. Using `which()`, the result is given as just the row number in `riverquality$Temp`. Using “`riverquality$Temp<3`” prints the entire column as a set of TRUE, FALSE or NA responses for each cell, and involves counting which value meets the TRUE criterion.
Figure 2.2: Value plot of the variable AnDet from the riverquality data frame

Subscripts are the key to navigating around a data frame, matrix or array and are just x, y (and z in case of arrays) co-ordinates, as in [row no., column no.]. To access the value in row 35 within the column Temp we enter:

```
riverquality[35, "Temp"]
```

```
[1] 9.691837
```

To edit the value we just assign it a new value:

```
#assigning a value using subscripts
riverquality[35,"Temp"]<-4
riverquality[35,"Temp"]
```
If using the column name it is IMPORTANT enclose the text in speech marks (""), otherwise R will not recognise what you are trying to do and will report either an error message or NULL.

There is another way of viewing the entire row (and the exact value we are looking for), using `fix()` to launch the data editor:

```r
#using fix to edit a data frame
fix(riverquality)
```

Scroll down the resulting spreadsheet to row 35 and we see the same result as if using subscripts. To edit a cell, select the cell and enter the new value. The data editor is also useful for scanning your data and changing any incorrect values that are not obvious from using plots.

Plots can be useful for identifying any outlier which may need investigation and may be incorrectly entered in the original spreadsheet. In the riverquality data set this can be seen by plotting Chloride. The value is 417 and appears to be anomalous. The original data set actually indicates that this is not the case but it is important to check from your original record.

We need to check each variable. Work through the variables plotting each one to find any outliers. A faster way would be to plot several graphs onto one page. This is possible using `par(mfrow=c(no. of rows, no. of columns))`. It should be noted that the more graphs that are put on the page the less detail is visible.

```r
#multiple plots per page
par(mfrow=c(1,3))
with(riverquality,plot(Temp))
with(riverquality,plot(pH))
with(riverquality,plot(Cond))
```
Alternatively, pair plots are useful to initially explore data and can identify possible incorrectly entered values. Again it will be easier to limit the number of graphs per page:

```r
#using pair plots in data exploration
pairs(riverquality[,5:12],upper.panel=NULL)
pairs(riverquality[,13:18],upper.panel=NULL)
```

The subscripts here denote the column numbers to include (we know that the first five columns are factors not variables. It is quicker to look up the column numbers (using `names(riverquality)`) than typing out each column name, which would look like this:

```r
pairs(riverquality[,c("Temp","pH","Cond","SS","Ash","DO","BOD","Amm")],upper.panel=NULL)
```

Note that doing it this way, we have to use `c()` to allow for the list of text names. The result should look like figure 2.3. Pairs plots show where potential sources of further investigation could be carried out and provide context for some potential outliers. Chlorophyll and Chloride show the most distinct outliers (though the original data set indicates these values are correct).
Figure 2.3: Pairs plot of the first 8 variables from the river quality data frame
Subscripts can also be used if we want to extract specific data from a large dataset. For instance we may only be interested in a specific region of the UK, such as the Scottish rivers, rows 482 to 668. We can extract these rows, dropping the first column, ‘Region’, and creating a new object using just the subscripts:

```r
cotriver<-riverquality[482:668,2:18]
names(cotriver)
```

Note: the new subset will contain the original case numbers, ie 482 through 668, retained as ‘row.names’.

Alternatively we may want to compare river quality by country rather than just by region. To do so, we need to create a new factor within the riverquality data frame using `ifelse(if this, replace with, else):

```r
# Using ifelse when creating a new factor variable
riverquality$country<-ifelse(riverquality$Region=="EA Wales","Wales","England")

# Using pattern matching to conditionally assign values
riverquality$country<-ifelse(grepl("SEPA.*",riverquality$Region),"Scotland",riverquality$country)

... and to ensure that country is a factor, finish with:

```r
riverquality$country<-as.factor(riverquality$country)
levels(riverquality$country)
```

[1] "England"  "Scotland" "Wales"

You may wish to save your factor as a csv file so it may be viewed or manipulated in Excel:

```r
# Save your output as a csv file
write.table(riverquality,file="M:\RDM\Arrrr\riverquality.csv",row.names=FALSE,sep=",")
```
Now if we want to extract data specific to any country we can use `which`:

```r
scotriver <- riverquality[which(riverquality$country == "Scotland"),]
```

You will discover if you look at the levels of `scotriver$River` that all of the labels from `riverquality$River` are shown (e.g. Mersey, Thames etc). This is a bit untidy and will make any summary tables look untidy and full of “NA”. To avoid this problem, remove the extraneous labels using `droplevels()`:

```r
scotriver
levels(scotriver$River)
scotriver <- droplevels(scotriver)
levels(scotriver$River)
```

```
[1] "CLYDE - GLASGOW GREEN"  "
[2] "DEE - GLENLOCHAR GAUGING STATION"  "
[3] "LOCHY - A830 ROAD BRIDGE"
[4] "SPEY - FOCHABERS"  "
[5] "TAY - PERTH (QUEENS BRIDGE)"  "
[6] "TWEED - NORHAM"  
```

**Working with arrays and matrices**

Subscripts are most useful when navigating around arrays and matrices. Here we will use the array and matrices created in Exercise 1. First we will label the rows and columns of both using a variety of `dimnames()`. We need to first specify which dimensions we are naming and then the names we are applying to them. Rows are always the first dimension and columns the second (in the case of arrays the separate tables form the third dimension).

```r
dimnames(matexample) <-
list(paste("site", 1:3, sep = "."),
paste(c("Quality", "No. Species", "Area")))
matexample
```
<table>
<thead>
<tr>
<th>site</th>
<th>Quality</th>
<th>No. Species</th>
<th>Area</th>
</tr>
</thead>
<tbody>
<tr>
<td>site.1</td>
<td>1</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>site.2</td>
<td>2</td>
<td>5</td>
<td>8</td>
</tr>
<tr>
<td>site.3</td>
<td>3</td>
<td>6</td>
<td>9</td>
</tr>
</tbody>
</table>

The use of the `paste()` function achieves the same result as if we had entered the names as below:

```r
> dimnames(matexample)<-
list(c("site.1","site.2","site.3"),c("Quality","No.Species","Area"))
```

Alternatively we can use `colnames()` and `rownames()` to specify the column names and row names of the matrix. For the columns we have to create a group containing the names before assigning the names to the columns:

```r
QNA<-c("Quality", "No.Species", "Area")
colnames(matexample)<-QNA
rownames(matexample)<-
rownames(matexample,do.NULL=FALSE,prefix= "site ")
```

or

```r
rownames(matexample)<-paste("site",1:3,sep=".")
```

N.B. If copying and pasting code from a word processor, you may get an error message “Error: unexpected input in..” due to the slight difference in the speech marks. For example speech marks ("") generated by MS Word have a slight angle to them (hex codes 201C and 201D) whereas in R they have to be vertical ("", hex code 22).

Labelling the dimensions of an array uses the same method as labelling a matrix. In this instance it is easier to create the labels prior to use. We'll use the array created in Exercise 1 (Arrayexample) with the data representing the number of animal and plant species in sections of three different rivers:
#labelling an Array

riverex<-'paste("River.",1:3,sep="")
sectionex<-'c("upper","upper middle","lower middle","lower")
biotaex<-'c("no.fauna sp","no.flora sp")
dimnames(Arrayexample)<-'list(sectionex,biotaex,riverex)

The result should be similar to that shown in figure 2.4.

Naming rows and columns will make it a bit easier to get to grips with using subscripts to navigate around matrices and arrays. We'll start with the data in matexample. We may be interested in only the data in site.1 (the first row):

```
matexample[1,]
>       Quality  No. Species  Area
> 1 4 7
```

Or the number of species for each site (column two):

```
matexample[,2]
> site.1 site.2 site.3
> 4 5 6
```

or
```
matexample[,"No.Species "]
>site.1 site.2 site.3
>4 5 6
```

As with adding new variables or data to an existing data frame, it is possible to use a similar method to add columns or rows to matrices (arrays are slightly problematic due to the greater number of dimensions). In this instance we will use `cbind()` to add a column (a column containing the total for each...
row) and `rbind()` (to add a row). We'll start with matexample, adding an additional site and a new variable:

```r
#adding a new row and column to a matrix
site.4 <- c(4, 7, 10)
matexample <- rbind(matexample, site.4)
Total <- apply(matexample, 1, sum)
matexample <- cbind(matexample, Total)
matexample
```

<table>
<thead>
<tr>
<th>Quality</th>
<th>No.Species</th>
<th>Area</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>site.1</td>
<td>1</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>site.2</td>
<td>2</td>
<td>5</td>
<td>8</td>
</tr>
<tr>
<td>site.3</td>
<td>3</td>
<td>6</td>
<td>9</td>
</tr>
<tr>
<td>site.4</td>
<td>4</td>
<td>7</td>
<td>10</td>
</tr>
</tbody>
</table>
Figure 2.4: Labelling the different dimensions within an array using Arrayexample.
Arrays are more problematic when it comes to adding data due to having more than two dimensions and therefore, if wishing to keep the data within an array, a different approach is taken. We will work with Arrayexample. In the first instance we want to add an extra river (an additional third dimension):

```r
riverarray<-c(as.vector(Arrayexample),rpois(8,2))
dim(riverarray)<-c(4,2,4)
```

We will need to add the dimension names again, as when coercing Arrayexample into a vector all the names were stripped. We can use the same names as previously used for Arrayexample with only two important alterations:

```r
riverex<-paste("River.",1:4,sep="")
sectionex<-c("upper","upper middle","lower middle","lower")
biotaex<-c("no.fauna sp","no.flora sp")
dimnames(riverarray)<-list(sectionex,biotaex,riverex)
```

Compared to the earlier example, `riverex` has been expanded (to 1:4) and `dimnames` now takes `riverarray` rather than `Arrayexample` as its argument.

The main difficulty comes with adding new rows or columns (the first and second dimensions). The easiest method would be to create a completely new array with the new data. Otherwise it is possible to use `fix()` to add data, paying careful attention to where the data should be added:

```r
fix(riverarray)
```

...will produce a window similar to that shown in figure 2.5. N.B. when you earlier used `fix()` to open a matrix, it opened a spreadsheet-like editor, whereas `riverarray` opens in a notepad-like editor because it is not a matrix. It is very important to know where in the order of the existing data items the new data items need to be added. In figure 2.5 the new data has been added. To
understand how to read the data, the array in vector form is deciphered as: first
four numbers = upper to lower in column 1 (river 1); next four = upper to lower
in column 2 (river 1); next four = upper to lower in column 1 (river 2); next four
= upper to lower in column 2 (river 2) and so on. For ease, 0s have been added,
to create a new row called “source” (as shown under .Dimnames). We also need
to change the number of rows (under .Dim) from 4L to 5L (remember rows come
first when setting the number of dimensions).
Now when we close the window (clicking “yes” when asked if we want to save
the changes) we should get no error message. If we do it means that either we
have not added enough data or not set the correct number of dimensions and it
just means going back to ensure the correct values are added in the correct place.
We can add a new column using the same process. Take some time now to work
on this and experiment.
Figure 2.5: Adding columns and rows to an array. Example used is riverarray