3

Data Input

You can get numbers into R through the keyboard, from the clipboard or from an external file. For a single variable of up to 10 numbers or so, it is probably quickest to type the numbers at the command line, using the *concatenate* function c like this:

```
y <- c (6,7,3,4,8,5,6,2)
```

For intermediate sized variables, you might want to enter data from the keyboard using the scan function. For larger data sets, and certainly for sets with several variables, you should make a dataframe in Excel and read it into R using read.table (p. 98).

The scan Function

This is the function to use if you want to type (or paste) a few numbers into a vector from the keyboard.

```
x<-scan()

1:
```

At the `1:` prompt type your first number, then press the Enter key. When the `2:` prompt appears, type in your second number and press Enter, and so on. When you have put in all the numbers you need (suppose there are eight of them) then simply press the Enter key at the `9:` prompt.

```
x<-scan()

1: 6
2: 7
3: 3
4: 4
5: 8
6: 5
7: 6
8: 2
9:
Read 8 items
```

You can also use scan to paste in groups of numbers from the clipboard. In Excel, highlight the column of numbers you want, then type Ctrl+C (the accelerator keys for Copy). Now go back into R. At the `1:` prompt just type Ctrl+V (the accelerator keys for Paste) and the numbers will be scanned into the named variable (*x* in this example). You can then paste in another set of numbers, or press Return to complete data entry. If you try to read in a group of numbers from a *row* of cells in Excel, the characters will be pasted into a single multi-digit number (definitely *not* what is likely to have been intended). So, if you are going to paste numbers from Excel, make sure the numbers are in columns, not in rows, in the spreadsheet. Use Edit / Paste Special / Transpose in Excel to turn a row into a column if necessary.

Data Input from Files

You can read data from a file using scan (see p. 102) but read.table is much more user-friendly. The read.table function reads a file in table format and automatically creates a dataframe from it, with cases corresponding to rows (lines) and variables to columns (fields) in the file (see p. 107). Much the simplest way to proceed is always to make your dataframe as a spreadsheet in Excel, and always to save it as a tab-delimited text file. That way you will always use read.table for data input, and you will avoid many of the most irritating problems that people encounter when using other input formats.

Saving the File from Excel

Once you have made your dataframe in Excel and corrected all the inevitable data-entry and spelling errors, then you need to save the dataframe in a file format that can be read by R. Much the simplest way is to save all your dataframes from Excel as tab-delimited text files: File/Save As . . . / then from the 'Save as type' options choose 'Text (Tab delimited)'. There is no need to add a suffix, because Excel will automatically add '.txt' to your file name. This file can then be read into R directly as a dataframe, using the read.table function like this:

```
data<-read.table("c:\\temp\\regression.txt",header=T)
```

Common Errors when Using read.table

It is important to note that read.table would fail if there were any spaces in any of the variable names in row 1 of the dataframe (the header row, see p. 107), such as Field Name, Soil pH or Worm Density, or between any of the words within the same factor level (as in many of the field names). You should replace all these spaces by dots '.' before saving the dataframe in Excel (use Edit/Replace with " " replaced by "."). Now the dataframe can be read into R. There are three things to remember:

- The whole path and file name needs to be enclosed in double quotes: "c:\\abc.txt".
- header=T says that the first row contains the variable names.
- Always use double backslash \\ rather than \ in the file path definition.

The commonest cause of failure is that the number of variable names (characters strings in row 1) does not match the number of columns of information. In turn, the commonest cause of this is that you have blank spaces in your variable names:

```
state name  population  home ownership  cars  insurance
```

This is wrong because R expects seven columns of numbers when there are only five. Replace the spaces within the names by dots and it will work fine:

```
state.name  population  home.ownership  cars  insurance
```

The next most common cause of failure is that the data file contains blank spaces where there are missing values. Replace these blanks with NA in Excel (or use a different separator symbol: see below).

Finally, there can be problems when you are trying to read variables that consist of character strings containing blank spaces (as in files containing place names). You can use read.table so long as you export the file from Excel using commas to separate the fields, and you tell read.table that the separators are commas using sep=","

```
map<-read.table("c:\\temp\\bowens.csv",header=T,sep=",")
```

but it is quicker and easier to use read.csv in this case(and there is no need for header=T)

```
map<-read.csv("c:\\temp\\bowens.csv")
```

If you are tired of writing header=T in all your read.table functions, then switch to

```
read.delim("c:\\temp\\file.txt")
```

or write your own function

```
rt<-function(x) read.delim(x)
```

then use the function rt to read data-table files like this:

```
rt("c:\\temp\\regression.txt")
```

Better yet, remove the need to enter the drive and directory or the file suffix:

```
rt<-function(x) read.delim(paste("c:\\temp\\",x,".txt",sep=""))
rt("regression")
```

Browsing to Find Files

The R function for this is file.choose(). Here it is in action with read.table:

```
data<-read.table(file.choose(),header=T)
```

Once you click on your selected file this is read into the dataframe called data.

Separators and Decimal Points

The default field separator character in read.table is sep="". This separator is white space, which is produced by one or more spaces, one or more tabs \t, one or more

newlines \n, or one or more carriage returns. If you do have a different separator between the variables sharing the same line (i.e. other than a tab within a .txt file) then there may well be a special read function for your case. Note that these all have the sensible default that header=TRUE (the first row contains the variable names): for comma-separated fields use read.csv("c:\\temp\\file.txt"), for semicolon separated fields read.csv2("c:\\temp\\file.txt"), and for decimal points as a comma read.delim2("c:\\temp\\file.txt"). You would use comma or semicolon separators if you had character variables that might contain one or more blanks (e.g. country names like 'United Kingdom' or 'United States of America').

If you want to specify row.names then one of the columns of the dataframe must be a vector of unique row names. This can be a single number giving the column of the table which contains the row names, or character string giving the variable name of the table column containing the row names (see p. 123). Otherwise if row.names is missing, the rows are numbered.

The default behaviour of read.table is to convert character variables into factors. If you do *not* want this to happen (you want to keep a variable as a character vector) then use as.is to specify the columns that should not be converted to factors:

```
murder<-read.table("c:\\temp\\murders.txt",header=T,as.is="region"); attach(murder)
```

We use the attach function so that the variables inside a dataframe can be accessed directly by name. Technically, this means that the database is attached to the R search path, so that the database is searched by R when evaluating a variable.

```
table(region)
```

```
region
North.Central  Northeast  South  West
           12          9     16    13
```

If we had not attached a dataframe, then we would have had to specify the name of the dataframe first like this:

```
table(murder$region)
```

The following warning will be produced if your attach function causes a duplication of one or more names:

```
The following object(s) are masked _by_ .GlobalEnv:
        murder
```

The reason in the present case is that we have created a dataframe called murder and attached a variable which is also called murder. This ambiguity might cause difficulties later. The commonest cause of this problem occurs with simple variable names like x and y. It is very easy to end up with multiple variables of the same name within a single session that mean totally different things. The warning after using attach should alert you to the possibility of such problems. If the vectors sharing the same name are of different lengths, then R is likely to stop you before you do anything too silly, but if the vectors are the same length then you run the serious risk of fitting the wrong explanatory variable (e.g. fitting the wrong one from two vectors both called x) or having the wrong response variable (e.g. from two vectors both called y). The moral is:

- use longer, more self-explanatory variable names;
- do not calculate variables with the same name as a variables inside a dataframe;
- always detach dataframes once you are finished using them;
- remove calculated variables once you are finished with them (rm; see p. 8).

The best practice, however, is not to use attach in the first place, but to use functions like with instead (see p. 18). If you get into a real tangle, it is often easiest to quit R and start another R session. To check that region is not a factor, write:

```
is.factor(region)
```

```
[1] FALSE
```

Input and Output Formats

Formatting is controlled using **escape sequences**, typically within double quotes:

\n	newline
\r	carriage return
\t	tab character
\b	backspace
\a	bell
\f	form feed
\v	vertical tab

Setting the Working Directory

You do not have to type the drive name and folder name every time you want to read or write a file, so if you use the same path frequently it is sensible to set the working directory using the setwd function:

```
setwd("c:\\temp")
. . .
. . .
read.table("daphnia.txt",header=T)
```

If you want to find out the name of the current working directory, use getwd():

```
getwd()
```

```
[1] "c:/temp"
```

Checking Files from the Command Line

It can be useful to check whether a given filename exists in the path where you think it should be. The function is file.exists and is used like this:

```
file.exists("c:\\temp\\Decay.txt")
```

```
[1] TRUE
```

For more on file handling, see ?files.

Reading Dates and Times from Files

You need to be very careful when dealing with dates and times in any sort of computing. R has a particularly robust system for working with dates and times, which is explained in detail on p. 89. Typically, you will read dates and times as character strings, then convert them into dates and/or times within R.

Built-in Data Files

There are many built-in data sets within the base package of R. You can see their names by typing

```
data()
```

You can read the documentation for a particular data set with the usual query:

```
?lynx
```

Many of the contributed packages contain data sets, and you can view their names using the try function. This evaluates an expression and traps any errors that occur during the evaluation. The try function establishes a handler for errors that uses the default error handling protocol:

```
try(data(package="spatstat"));Sys.sleep(3)
try(data(package="spdep"));Sys.sleep(3)
try(data(package="MASS"))
```

Built-in data files can be attached in the normal way; then the variables within them accessed by their names:

```
attach(OrchardSprays)
decrease
```

Reading Data from Files with Non-standard Formats Using scan

The scan function is very flexible, but as a consequence of this, it is much harder to use than read.table. This example uses the US murder data. The filename comes first, in the usual format (enclosed in double quotes and using paired backslashes to separate the drive name from the folder name and the folder name from the file name). Then comes skip=1 because the first line of the file contains the variable names (as indicated by header=T in a read.table function). Next comes what, which is a list of length the number of variables (the number of columns to be read; 4 in this case) specifying their type (character "" in this case):

```
murders<-scan("c:\\temp\\murders.txt", skip=1, what=list("","","",""))
```

```
Read 50 records
```

The object produced by scan is a list rather than a dataframe as you can see from

```
class(murders)
```

```
[1] "list"
```

It is simple to convert the list to a dataframe using the as.data.frame function

```
murder.frame<-as.data.frame(murders)
```

You are likely to want to use the variables names from the file as variable names in the dataframe. To do this, read just the first line of the file using scan with nlines=1:

```
murder.names<-
scan("c:\\temp\\murders.txt",nlines=1,what="character",quiet=T)
murder.names
```

```
[1] "state" "population" "murder" "region"
```

Note the use of quiet=T to switch off the report of how many records were read. Now give these names to the columns of the dataframe

```
names(murder.frame)<-murder.names
```

Finally, convert columns 2 and 3 of the dataframe from factors to numbers:

```
murder.frame[,2]<-as.numeric(murder.frame[,2])
murder.frame[,3]<-as.numeric(murder.frame[,3])
summary(murder.frame)
```

```
   state          population      murder            region
Alabama     : 1  Min.    : 1.00  Min.   : 1.00  North.Central :12
Alaska      : 1  1st Qu. :13.25  1st Qu.:11.25  Northeast     : 9
Arizona     : 1  Median  :25.50  Median :22.50  South         :16
Arkansas    : 1  Mean    :25.50  Mean   :22.10  West          :13
California  : 1  3rd Qu. :37.75  3rd Qu.:32.75
Colorado    : 1  Max.    :50.00  Max.   :44.00
(Other)     :44
```

You can see why people prefer to use read.table for this sort of data file:

```
murders<-read.table("c:\\temp\\murders.txt",header=T)
summary(murders)
```

```
   state          population      murder            region
Alabama     : 1  Min.    :  365  Min.    : 1.400  North.Central :12
Alaska      : 1  1st Qu. : 1080  1st Qu. : 4.350  Northeast     : 9
Arizona     : 1  Median  : 2839  Median  : 6.850  South         :16
Arkansas    : 1  Mean    : 4246  Mean    : 7.378  West          :13
California  : 1  3rd Qu. : 4969  3rd Qu. :10.675
Colorado    : 1  Max.    :21198  Max.    :15.100
(Other)     :44
```

Note, however, that the scan function is quicker than read.table for input of large (numeric only) matrices.

Reading Files with Different Numbers of Values per Line

Here is a case where you might want to use scan because the data are not configured like a dataframe. The file rt.txt has different numbers of values per line (a neighbours file in spatial analysis, for example; see p. 769). In this example, the file contains five lines with 1, 2, 4, 2 and 1 numbers respectively: in general, you will need to find out the number of lines of data in the file by counting the number of end-of-line control character "\n" using the length function like this:

```
line.number<-length(scan("c:\\temp\\rt.txt",sep="\n"))
```

The trick is to combine the skip and nlines options within scan to read one line at a time, skipping no lines to read the first row, skipping one row to read the second line, and so on. Note that since the values are numbers we do not need to specify what:

```
(my.list<-sapply(0:(line.number-1),
   function(x) scan("c:\\temp\\rt.txt",skip=x,nlines=1,quiet=T)))
```

```
[[1]]
[1] 138

[[2]]
[1] 27 44

[[3]]
[1] 19 20 345 48

[[4]]
[1] 115 23 66

[[5]]
[1] 59
```

The scan function has produced a list of vectors, each of a different length. You might want to know the number of numbers in each row, using length with lapply like this:

```
unlist(lapply(my.list,length))
```

```
[1] 1 2 4 2 1
```

Alternatively, you might want to create a vector containing the *last element* from each row:

```
unlist(lapply(1:length(my.list), function(i) my.list[[i]][length(my.list[[i]])]))
```

```
[1] 138 44 48 2366 59
```

The readLines Function

In some cases you might want to read each line from a file separately. The argument n=-1 means read to the end of the file. Let's try it out with the murders data (p. 100):

```
readLines("c:\\temp\\murders.txt",n=-1)
```

This produces the rather curious object of class = "character":

```
[1] "state\tpopulation\tmurder\tregion"  "Alabama\t3615\t15.1\tSouth"
[3] "Alaska\t365\t11.3\tWest"            "Arizona\t2212\t7.8\tWest"
```

```
.......
........
[49] "West.Virginia\t1799\t6.7\tSouth"    "Wisconsin\t4589\t3\tNorth.Central"
[51] "Wyoming\t376\t6.9\tWest"
```

Each line has been converted into a single character string. Line [1] contains the four variable names (see above) separated by tab characters \t. Line [2] contains the first row of data for murders in Alabama, while row [51] contains the last row of data for murders in Wyoming. You can use the string-splitting function strsplit to tease apart the elements of the string (say, for the Wyoming data [51]):

```
mo<-readLines("c:\\temp\\murders.txt",n=-1)
strsplit(mo[51],"\t")
```

```
[[1]]
[1] "Wyoming""376" "6.9" "West"
```

You would probably want 376 and 6.9 as numeric rather than character objects:

```
as.numeric(unlist(strsplit(mo[51],"\t")))
```

```
[1]  NA  376.0  6.9  NA
Warning message:
NAs  introduced by coercion
```

where the two names Wyoming and West have been coerced to NA, or

```
as.vector(na.omit(as.numeric(unlist(strsplit(mo[51],"\t")))))
```

```
[1] 376.0 6.9
Warning message:
NAs introduced by coercion
```

to get the numbers on their own. Here is how to extract the two numeric variables (murder = mur and population = pop) from this object using sapply:

```
mv<-sapply(2:51,function(i)
            as.vector(na.omit(as.numeric(unlist(strsplit(mo[i],"\t"))))))
pop<-mv[1,]
mur<-mv[2,]
```

and here is how to get character vectors of the state names and regions (the first and fourth elements of each row of the list called ms, called sta and reg, respectively):

```
ms<-sapply(2:51,function(i) strsplit(mo[i],"\t"))
texts<-unlist(lapply(1:50,function(i) ms[[i]][c(1,4)]))
sta<-texts[seq(1,99,2)]
reg<- texts[seq(2,100,2)]
```

Finally, we can convert all the information from readLines into a data.frame

```
data.frame(sta,pop,mur,reg)
```

```
           sta   pop   mur            reg
1      Alabama  3615  15.1          South
2       Alaska   365  11.3           West
...
49   Wisconsin  4589   3.0  North.Central
50     Wyoming   376   6.9           West
```

This could all have been achieved in a single line with read.table (see above), and the readLines function is much more useful when the rows in the file contain different numbers of entries. Here is the simple example from p. 104 using readLines instead of scan:

```
rlines<-readLines("c:\\temp\\rt.txt")
split.lines<-strsplit(rlines,"\t")
new<-sapply(1:5,function(i) as.vector(na.omit(as.numeric(split.lines[[i]]))))
new
```

```
[[1]]
[1] 138

[[2]]
[1] 27 44

[[3]]
[1] 19 20 345 48

[[4]]
[1] 115 2366

[[5]]
[1] 59
```

The key features of this procedure are the removal of the tabs (\t) and the separation of the values with each row with the strsplit function, the conversion of the characters to numbers, and the removal of the NAs which are introduced by default by the as.numeric function. I think that scan (p. 104) is more intuitive in such a case.