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 externally (e.g. in a spreadsheet) and read it into R using `read.table` (p. 139).

3.1 Data input from the keyboard

The `scan` function is useful if you want to type (or paste) a few numbers into a vector called x 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.

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

Read 8 items
```

The R Book, Second Edition. Michael J. Crawley.

You can also use `scan` to paste in columns of numbers from the Clipboard. In the spreadsheet, 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. If necessary, use Edit/Paste Special/Transpose in Excel to turn a row into a column before copying the numbers.

3.2 Data input from files

The easiest way to get data into R is to make your data into the shape of a dataframe before trying to read it into R. As explained in detail in Chapter 4, you should put all of the values of each variable into a single column, and put the name of the variable in row 1 (called the 'header' row). You will sometimes see the rows and the columns of the dataframe referred to as *cases* and *fields* respectively. In our terminology, the fields are the *variables* and the cases are *rows*.

Where you have text strings containing blanks (e.g. place names like 'New Brighton') then `read.table` is no good because it will think that 'New' is the value of one variable and 'Brighton' is another, causing the input to fail because the number of data items does not match the number of columns. In such cases, use a comma to separate the fields, and read the data from file using the function `read.csv` (the suffix `.csv` stands for 'comma separated values'). The function `read.csv2` is the variant for countries where a comma is used as the decimal point: in this case, a semicolon is the field separator.

3.2.1 The working directory

It is useful to have a working directory, because this makes accessing and writing files so much easier, since you do not need to type the (potentially long) path name each time. For instance, we can simplify data entry by using setwd to set the working directory to `"c:\\temp"`:

```
setwd("c:\\temp")
```

To find out the name of the current working directory, use `getwd()` like this:

```
getwd()
```

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

If you have not changed the working directory, then you will see something like this:

```
getwd()
```

```
[1] "C:/Documents and Settings/mjcraw/My Documents"
```

which is the working directory assumed by R when you start a new session. If you plan to switch back to the default working directory during a session, then it is sensible to save the long default path before you change it, like this:

```
mine<-getwd()
setwd("c:\\temp")
...
setwd(mine)
```

If you want to view file names from R, then use the `dir` function like this:

```
dir("c:\\temp")
```

To pick a file from a directory using the browser, invoke the `file.choose` function. This is most used in association with `read.table` if you have forgotten the name or the location of the file you want to read into a dataframe:

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

Just click on the file you want to read, and the contents will be assigned to the frame called data.

3.2.2 Data input using `read.table`

Here is an example of the standard means of data entry, creating a dataframe within R using `read.table`:

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

```
  year wheat barley oats rye corn
1 1980   5.9    4.4  4.1 3.8  4.4
2 1981   5.8    4.4  4.3 3.7  4.1
3 1982   6.2    4.9  4.4 4.1  4.0
4 1983   6.4    4.7  4.3 3.7  4.1
5 1984   7.7    5.6  4.9 4.7  4.7
6 1985   6.3    5.0  4.6 4.6  4.3
```

You can save time by using `read.delim`, because then you can omit `header=T`:

```
data <- read.delim("yield.txt")
```

Or you could go the whole way down the labour-saving route and write your own text-minimizing function, which uses the `paste` function to add the suffix `.txt` as well as shortening the function name from `read.table` to `rt` like this:

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

Then simply type:

```
data <- rt("yields")
```

3.2.3 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. 161), 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 (`T` stands for `TRUE`).
- Always use double \\ rather than \ in the file path definition.

The most common 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 left 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 your spreadsheet before reading the file into R.

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 the spreadsheet using commas to separate the fields, and you tell `read.table` that the separators are commas using `sep=","` (to override the default blanks or tabs `(\t)`:

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

However, it is quicker and easier to use `read.csv` in this case (see below).

3.2.4 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 all the alternatives to `read.table` have the sensible default that `header=TRUE` (the first row contains the variable names):

- for comma-separated fields use `read.csv("c:\\temp\\file.csv")`;
- for semicolon-separated fields `read.csv2("c:\\temp\\file.csv")`;
- for tab-delimited fields with decimal points as a commas, use `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. 176). Otherwise, if `row.names` is missing, rows numbers are generated automatically on the left of the dataframe.

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")
```

3.2.5 Data input directly from the web

You will typically use `read.table` to read data from a file, but the function also works for complete URLs. In computing, URL stands for 'universal resource locator', and is a specific character string that constitutes a

reference to an Internet resource, combining domain names with file path syntax, where forward slashes are used to separate folder and file names:

```
data2 <- read.table
("http://www.bio.ic.ac.uk/research/mjcraw/therbook/data/cancer.txt",
  header=T)
head(data2)
```

```
  death treatment status
1     4      DrugA      1
2    26      DrugA      1
3     2      DrugA      1
4    25      DrugA      1
5     7      DrugA      1
6     6      DrugA      0
```

3.3 Input from files using `scan`

For dataframes, `read.table` is superb. But look what happens when you try to use `read.table` with a more complicated file structure:

```
read.table("c:\\temp\\rt.txt")
```

```
Error in scan(file, what, nmax, sep, dec, quote, skip, nlines, na.strings
   line 1 did not have 4 elements
```

It simply cannot cope with lines having different numbers of fields. However, `scan` and `readLines` come into their own with these complicated, non-standard files.

The `scan` function reads data into a list when it is used to read from a file. It is much less friendly for reading dataframes than `read.table`, but it is substantially more flexible for tricky or non-standard files.

By default, `scan` assumes that you are inputting double precision numbers. If not, then you need to use the `what` argument to explain what exactly you are inputting (e.g. 'character', 'logical', 'integer', 'complex' or 'list'). The file name is interpreted relative to the current working directory (given by `getwd()`), unless you specify an absolute path. If `what` is itself a list, then `scan` assumes that the *lines* of the data file are records, each containing one or more items and the number of *fields* on that line is given by `length(what)`.

By default, `scan` expects to read space-delimited or tab-delimited input fields. If your file has separators other than blank spaces or tab markers (`\t`), then you can specify the separator option (e.g. `sep=","`) to specify the character which delimits fields. A field is always delimited by an end-of-line marker unless it is quoted. If `sep` is the default (`""`), the character `\` in a quoted string escapes the following character, so quotes may be included in the string by escaping them (`"she.said \"What!\".to.him"`). If `sep` is non-default, the fields may be quoted in the style of `.csv` files where separators inside quotes (`''` or `""`) are ignored and quotes may be put inside strings by doubling them. However, if `sep = "\n"` it is assumed by default that one wants to read entire lines verbatim.

With `scan`, you will often want to skip the header row (because this contains variable names rather than data). The option for this is `skip = 1` (you can specify any number of lines to be skipped before beginning to read data values). If a single record occupies more than one line of the input file then use the option `multi.line =TRUE`.

3.3.1 Reading a dataframe with `scan`

To illustrate the issues involved with `scan`, and to demonstrate why `read.table` is the preferred way of reading data into dataframes, we use `scan` to input the `worms` dataframe (for details, see Chapter 4).

We want to skip the first row because that is a header containing the variable names, so we specify `skip = 1`. There are seven columns of data, so we specify seven fields of character variables `""` in the list supplied to `what`:

```
scan("t:\\data\\worms.txt",skip=1,what=as.list(rep("",7)))
```

```
Read 20 records
[[1]]
 [1] "Nashs.Field"       "Silwood.Bottom"    "Nursery.Field"
 [4] "Rush.Meadow"       "Gunness.Thicket"   "Oak.Mead"
 [7] "Church.Field"      "Ashurst"           "The.Orchard"
[10] "Rookery.Slope"     "Garden.Wood"       "North.Gravel"
[13] "South.Gravel"      "Observatory.Ridge" "Pond.Field"
[16] "Water.Meadow"      "Cheapside"         "Pound.Hill"
[19] "Gravel.Pit"        "Farm.Wood"

[[2]]
 [1] "3.6" "5.1" "2.8" "2.4" "3.8" "3.1" "3.5" "2.1" "1.9" "1.5" "2.9" "3.3"
[13] "3.7" "1.8" "4.1" "3.9" "2.2" "4.4" "2.9" "0.8"

[[3]]
 [1] "11" "2"  "3"  "5"  "0"  "2"  "3"  "0"  "0"  "4"  "10" "1"  "2"  "6"  "0"
[16] "0"  "8"  "2"  "1"  "10"

[[4]]
 [1] "Grassland" "Arable"    "Grassland" "Meadow"    "Scrub"     "Grassland"
 [7] "Grassland" "Arable"    "Orchard"   "Grassland" "Scrub"     "Grassland"
[13] "Grassland" "Grassland" "Meadow"    "Meadow"    "Scrub"     "Arable"
[19] "Grassland" "Scrub"

[[5]]
 [1] "4.1" "5.2" "4.3" "4.9" "4.2" "3.9" "4.2" "4.8" "5.7" "5"   "5.2" "4.1"
[13] "4"   "3.8" "5"   "4.9" "4.7" "4.5" "3.5" "5.1"

[[6]]
 [1] "FALSE" "FALSE" "FALSE" "TRUE"  "FALSE" "FALSE" "FALSE" "FALSE" "FALSE"
[10] "TRUE"  "FALSE" "FALSE" "FALSE" "FALSE" "TRUE"  "TRUE"  "TRUE"  "FALSE"
[19] "FALSE" "TRUE"

[[7]]
 [1] "4" "7" "2" "5" "6" "2" "3" "4" "9" "7" "8" "1" "2" "0" "6" "8" "4" "5" "1"
[20] "3"
```

As you can see, `scan` has created a `list` of seven vectors of character string information. To convert this list into a dataframe, we use the `as.data.frame` function which turns the lists into columns in the dataframe (so long as the columns are all the same length):

```
data <-
as.data.frame(scan("t:\\data\\worms.txt",skip=1,what=as.list(rep("",7))))
```

In its present form, the variable names manufactured by `scan` are ridiculously long, so we need to replace them with the original variable names that are in the first row of the file. For this we can use `scan` again, but specify that we want to read only the first line, by specifying `nlines=1` and removing the `skip` option:

```
header <- unlist(scan("t:\\data\\worms.txt",nlines=1,what=as.list
  (rep("",7))))
```

Now, we replace the manufactured names by the correct variable names in `data`:

```
names(data)<-header
head(data)
```

```
       Field.Name Area Slope Vegetation Soil.pH Damp Worm.density
1     Nashs.Field  3.6    11  Grassland     4.1    F            4
2  Silwood.Bottom  5.1     2     Arable     5.2    F            7
3   Nursery.Field  2.8     3  Grassland     4.3    F            2
4     Rush.Meadow  2.4     5     Meadow     4.9    T            5
5 Gunness.Thicket  3.8     0      Scrub     4.2    F            6
6        Oak.Mead  3.1     2  Grassland     3.9    F            2
```

This has produced the right result, but you can see why you would want to use `read.table` rather than `scan` for reading dataframes.

3.3.2 Input from more complex file structures using `scan`

Here is an image of a file containing information on the identities of the neighbours of five individuals from a population: the first individual has one neighbour (number 138), the second individual has two neighbours (27 and 44), the third individual has four neighbours, and so on.

```
138
27     44
19     20    345    48
115    2366
59
```

See if you can figure out why the following three different readings of the same file `rt.txt` produce such different objects; the first has 10 items, the second 5 items and the third 20 items:

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

```
Read 10 items
 [1]  138   27   44   19   20  345   48  115 2366   59
```

```
scan("c:\\temp\\rt.txt",sep="\n")
```

```
Read 5 items
[1]       138      2744 192034548   1152366        59
```

```
scan("c:\\temp\\rt.txt",sep="\t")
```

```
Read 20 items
 [1]  138   NA   NA   NA   27   44   NA   NA   19   20  345   48  115
[14] 2366   NA   NA   59   NA   NA   NA
```

The only difference between the three calls to `scan` is in the specification of the separator. The first uses the default which is blanks or tabs (the 10 items are the 10 numbers that we are interested in, but information about their grouping has been lost). The second uses the new line `"\n"` control character as the separator (the contents of each of the five lines have been stripped out and trimmed to create meaningless numbers, except for 138 and 59 which were the only numbers on their respective lines). The third uses tabs `"\t"` as separators (we have no information on lines, but at least the numbers have retained their integrity, and missing values (`NA`) have been used to pad out each line to the same length, 4. To get the result we want we need to use

the information on the number of lines from method 2 and the information on the contents of each line from method 3. The first step is easy:

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

```
Read 5 items
[1] 5
```

So we have five lines of information in this file. To find the number of items per line were divide the total number of items

```
length(scan("c:\\temp\\rt.txt",sep="\t"))
```

```
Read 20 items
[1] 20
```

by the number of lines: 20 / 5 = 4. To extract the information on each line, we want to take a line at a time, and extract the missing values (i.e. remove the `NAs`). So, for line 1 this would be

```
scan("c:\\temp\\rt.txt",sep="\t")[1:4]
```

```
Read 20 items
[1] 138  NA  NA  NA
```

then, to remove the `NA` we use `na.omit`, to remove the `Read 20 items` we use `quiet=T` and to leave only the numerical value we use `as.numeric`:

```
as.numeric(na.omit(scan("c:\\temp\\rt.txt",sep="\t",quiet=T)[1:4]))
```

```
[1] 138
```

To complete the job, we need to apply this logic to each of the five lines in turn, to produce a list of vectors of variable lengths (1, 2, 4, 2 and 1):

```
sapply(1:5, function(i)
              as.numeric(na.omit(
              scan("c:\\temp\\rt.txt",sep="\t",quiet=T)[(4*i-3):
                (4*i)])))
```

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

[[2]]
[1]   27   44

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

[[4]]
[1]  115 2366

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

That was about as complicated a procedure as you are likely to encounter in reading information from a file. In hindsight, we might have created the data as a dataframe with missing values explicitly added to the rows that had less than four numbers. Then a single `read.table` statement would have been enough.

3.4 Reading data from a file using readLines

An alternative is to use readLines instead of scan. Here is a repeat of the example of reading the worms dataframe (above).

3.4.1 Input a dataframe using readLines

```
line<-readLines("c:\\temp\\worms.txt")
line
```

```
 [1] "Field.Name\tArea\tSlope\tVegetation\tSoil.pH\tDamp\tWorm.density"
 [2] "Nashs.Field\t3.6\t11\tGrassland\t4.1\tFALSE\t4"
 [3] "Silwood.Bottom\t5.1\t2\tArable\t5.2\tFALSE\t7"
 [4] "Nursery.Field\t2.8\t3\tGrassland\t4.3\tFALSE\t2"
 [5] "Rush.Meadow\t2.4\t5\tMeadow\t4.9\tTRUE\t5"
 [6] "Gunness.Thicket\t3.8\t0\tScrub\t4.2\tFALSE\t6"
 [7] "Oak.Mead\t3.1\t2\tGrassland\t3.9\tFALSE\t2"
 [8] "Church.Field\t3.5\t3\tGrassland\t4.2\tFALSE\t3"
 [9] "Ashurst\t2.1\t0\tArable\t4.8\tFALSE\t4"
[10] "The.Orchard\t1.9\t0\tOrchard\t5.7\tFALSE\t9"
[11] "Rookery.Slope\t1.5\t4\tGrassland\t5\tTRUE\t7"
[12] "Garden.Wood\t2.9\t10\tScrub\t5.2\tFALSE\t8"
[13] "North.Gravel\t3.3\t1\tGrassland\t4.1\tFALSE\t1"
[14] "South.Gravel\t3.7\t2\tGrassland\t4\tFALSE\t2"
[15] "Observatory.Ridge\t1.8\t6\tGrassland\t3.8\tFALSE\t0"
[16] "Pond.Field\t4.1\t0\tMeadow\t5\tTRUE\t6"
[17] "Water.Meadow\t3.9\t0\tMeadow\t4.9\tTRUE\t8"
[18] "Cheapside\t2.2\t8\tScrub\t4.7\tTRUE\t4"
[19] "Pound.Hill\t4.4\t2\tArable\t4.5\tFALSE\t5"
[20] "Gravel.Pit\t2.9\t1\tGrassland\t3.5\tFALSE\t1"
[21] "Farm.Wood\t0.8\t10\tScrub\t5.1\tTRUE\t3"
```

Each line has become a single character string. As you can see, we shall need to strip out all of the tab marks (\t), thereby separating the data entries and creating seven columns of information. The function for this is strsplit:

```
db<-strsplit(line,"\t")
db
```

```
[[1]]
[1] "Field.Name"     "Area" "Slope" "Vegetation" "Soil.pH" "Damp" "Worm.density"

[[2]]
[1] "Nashs.Field"     "3.6"    "11"  "Grassland"     "4.1"    "F"            "4"

[[3]]
[1] "Silwood.Bottom"  "5.1"     "2"     "Arable"     "5.2"    "F"            "7"

[[4]]
[1] "Nursery.Field"   "2.8"     "3"  "Grassland"     "4.3"    "F"            "2"
```

and so on for 21 elements of the list. The new challenge is to get this into a form that we can turn into a dataframe. The first issue is to get rid of the list structure using the `unlist` function:

```
bb<-unlist(db)
```

```
 [1] "Field.Name"          "Area"          "Slope"    "Vegetation"   "Soil.pH"   "Damp"
 [7] "Worm.density" "Nashs.Field"            "3.6"            "11" "Grassland"    "4.1"
[13] "F"                       "4" "Silwood.Bottom"           "5.1"         "2" "Arable"
[19] "5.2"                     "F"              "7" "Nursery.Field"         "2"      "3"
[25] "Grassland"           "4.3"              "F"   "Rush.Meadow"       "2.4"
```

and so on up to 147 items. We need to give this vector dimensions: the seven variable names come first, so the appropriate dimensionality is (7,21):

```
dim(bb)<-c(7,21)
bb
```

```
     [,1]           [,2]          [,3]             [,4]            [,5]          [,6]              [,7]
[1,] "Field.Name"   "Nashs.Field" "Silwood.Bottom" "Nursery.Field" "Rush.Meadow" "Gunness.Thicket" "Oak.Mead"
[2,] "Area"         "3.6"         "5.1"            "2.8"           "2.4"         "3.8"             "3.1"
[3,] "Slope"        "11"          "2"              "3"             "5"           "0"               "2"
[4,] "Vegetation"   "Grassland"   "Arable"         "Grassland"     "Meadow"      "Scrub"           "Grassland"
[5,] "Soil.pH"      "4.1"         "5.2"            "4.3"           "4.9"         "4.2"             "3.9"
[6,] "Damp"         "F"           "F"              "F"             "T"           "F"               "F"
[7,] "Worm.density" "4"           "7"              "2"             "5"           "6"               "2"
```

and so on for 21 rows. This is closer, but the rows and columns are the wrong way around. We need to transpose this object and drop the first row (because this is the header row containing the variable names):

```
t(bb)[-1,]
```

```
      [,1]                [,2]  [,3] [,4]         [,5]  [,6] [,7]
 [1,] "Nashs.Field"       "3.6" "11" "Grassland" "4.1" "F"  "4"
 [2,] "Silwood.Bottom"    "5.1" "2"  "Arable"    "5.2" "F"  "7"
 [3,] "Nursery.Field"     "2.8" "3"  "Grassland" "4.3" "F"  "2"
 [4,] "Rush.Meadow"       "2.4" "5"  "Meadow"    "4.9" "T"  "5"
 [5,] "Gunness.Thicket"   "3.8" "0"  "Scrub"     "4.2" "F"  "6"
 [6,] "Oak.Mead"          "3.1" "2"  "Grassland" "3.9" "F"  "2"
 [7,] "Church.Field"      "3.5" "3"  "Grassland" "4.2" "F"  "3"
 [8,] "Ashurst"           "2.1" "0"  "Arable"    "4.8" "F"  "4"
 [9,] "The.Orchard"       "1.9" "0"  "Orchard"   "5.7" "F"  "9"
[10,] "Rookery.Slope"     "1.5" "4"  "Grassland" "5"   "T"  "7"
[11,] "Garden.Wood"       "2.9" "10" "Scrub"     "5.2" "F"  "8"
[12,] "North.Gravel"      "3.3" "1"  "Grassland" "4.1" "F"  "1"
[13,] "South.Gravel"      "3.7" "2"  "Grassland" "4"   "F"  "2"
[14,] "Observatory.Ridge" "1.8" "6"  "Grassland" "3.8" "F"  "0"
[15,] "Pond.Field"        "4.1" "0"  "Meadow"    "5"   "T"  "6"
[16,] "Water.Meadow"      "3.9" "0"  "Meadow"    "4.9" "T"  "8"
[17,] "Cheapside"         "2.2" "8"  "Scrub"     "4.7" "T"  "4"
[18,] "Pound.Hill"        "4.4" "2"  "Arable"    "4.5" "F"  "5"
[19,] "Gravel.Pit"        "2.9" "1"  "Grassland" "3.5" "F"  "1"
[20,] "Farm.Wood"         "0.8" "10" "Scrub"     "5.1" "T"  "3"
```

That's more like it. Now the function `as.data.frame` should work:

```
frame<-as.data.frame(t(bb)[-1,])
head(frame)
```

```
                  V1  V2 V3         V4  V5 V6 V7
1        Nashs.Field 3.6 11 Grassland 4.1  F  4
2     Silwood.Bottom 5.1  2    Arable 5.2  F  7
3      Nursery.Field 2.8  3 Grassland 4.3  F  2
4        Rush.Meadow 2.4  5    Meadow 4.9  T  5
5    Gunness.Thicket 3.8  0     Scrub 4.2  F  6
6           Oak.Mead 3.1  2 Grassland 3.9  F  2
```

All we need to do now is add in the variable names: these are in the first row of the transpose of the object called `bb` (above):

```
names(frame)<-t(bb)[1,]
head(frame)
```

```
          Field.Name Area Slope Vegetation Soil.pH Damp  Worm.density
1        Nashs.Field  3.6    11  Grassland     4.1    F             4
2     Silwood.Bottom  5.1     2     Arable     5.2    F             7
3      Nursery.Field  2.8     3  Grassland     4.3    F             2
4        Rush.Meadow  2.4     5     Meadow     4.9    T             5
5    Gunness.Thicket  3.8     0      Scrub     4.2    F             6
```

The complexity of this procedure makes you appreciate just how useful the function `read.table` really is.

3.4.2 Reading non-standard files using `readLines`

Here is a repeat of the example of the neighbours file that we analysed using `scan` on p. 141:

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

```
[1] "138\t\t\t"       "27\t44\t\t"      "19\t20\t345\t48" "115\t2366\t\t"
[5] "59\t\t\t"
```

The `readLines` function had produced a vector of length 5 (one element for each row in the file), with the contents of each row reduced to a single character string (including the literal tab markers `\t`). We need to spit this up within the lines using `strsplit`: we split first on the tabs, then on the new lines in order to see the distinction:

```
strsplit(readLines("c:\\temp\\rt.txt"),"\t")
```

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

[[2]]
[1]  "27" "44"  ""

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

```
[[4]]
[1] "115" "2366"     ""

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

```
strsplit(readLines("c:\\temp\\rt.txt"),"\n")
```

```
[[1]]
[1] "138\t\t\t"

[[2]]
[1] "27\t44\t\t"

[[3]]
[1] "19\t20\t345\t48"

[[4]]
[1] "115\t2366\t\t"

[[5]]
[1] "59\t\t\t"
```

The split by tab markers is closest to what we want to achieve, so we shall work on that. First, turn the character strings into numbers:

```
rows<-lapply(strsplit(readLines("c:\\temp\\rt.txt"),"\t"),as.numeric)
rows
```

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

[[2]]
[1]   27   44  NA

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

[[4]]
[1]  115 2366  NA

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

Now all that we need to do is to remove the NAs from each of the vectors:

```
sapply(1:5, function(i) as.numeric(na.omit(rows[[i]])))
```

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

[[2]]
[1]   27   44

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

```
[[4]]
[1]  115 2366

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

Both `scan` and `readLines` were fiddly, but they got what we were looking for in the end. You will need a lot of practice before you appreciate when to use `scan` and when to use `readLines`. Writing lists to files is tricky (see p. 82 for an explanation of the options available).

3.5 Warnings when you `attach` the dataframe

Suppose that you read a file from data, then `attach` it:

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

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

```
attach(murder)

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 from this dataframe which is also called `murder`. This shows the cause of the problem: the dataframe name and the third variable name are identical:

```
head(murder)
```

```
       state population murder region
1    Alabama       3615   15.1  South
2     Alaska        365   11.3   West
3    Arizona       2212    7.8   West
4   Arkansas       2110   10.1  South
5 California      21198   10.3   West
6   Colorado       2541    6.8   West
```

This ambiguity might cause difficulties later. A much better plan is to give the dataframe a unique name (like `murders`, for instance). We use the `attach` function so that the variables inside a dataframe can be accessed directly by name. Technically, this means that the dataframe is attached to the R search path, so that the dataframe is searched by R when evaluating a variable. We could tabulate the numbers of murders by region, for instance:

```
table(region)
```

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

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

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