4

Dataframes

Learning how to handle your data, how to enter them into the computer, and how to read them into R are among the most important topics you will need to master. R handles data in objects known as dataframes. A **dataframe** is an object with rows and columns (a bit like a matrix). The rows contain different observations from your study, or measurements from your experiment (these are sometimes called *cases*). The columns contain the values of different variables (these are often called *fields*). The values in the body of a matrix can only be numbers, but the values in the body of a dataframe can be numbers, but they could also be text (e.g. the names of factor levels for categorical variables, like male or female in a variable called gender), they could be calendar dates (e.g. 23/5/04), or they could be logical variables (TRUE or FALSE). Here is a spreadsheet in the form of a dataframe with seven variables, the leftmost of which comprises the row names, and other variables are numeric (Area, Slope, Soil pH and Worm Density), categorical (Field Name and Vegetation) or logical (Damp is either true = T or false = F).

Field Name	Area	Slope	Vegetation	Soil pH	Damp	Worm Density
Nash's Field	3.6	11	Grassland	4.1	F	4
Silwood Bottom	5.1	2	Arable	5.2	F	7
Nursery Field	2.8	3	Grassland	4.3	F	2
Rush Meadow	2.4	5	Meadow	4.9	T	5
Gunness' Thicket	3.8	0	Scrub	4.2	F	6
Oak Mead	3.1	2	Grassland	3.9	F	2
Church Field	3.5	3	Grassland	4.2	F	3
Ashurst	2.1	0	Arable	4.8	F	4
The Orchard	1.9	0	Orchard	5.7	F	9
Rookery Slope	1.5	4	Grassland	5	T	7
Garden Wood	2.9	10	Scrub	5.2	F	8
North Gravel	3.3	1	Grassland	4.1	F	1
South Gravel	3.7	2	Grassland	4	F	2
Observatory Ridge	1.8	6	Grassland	3.8	F	0
Pond Field	4.1	0	Meadow	5	T	6
Water Meadow	3.9	0	Meadow	4.9	T	8
Cheapside	2.2	8	Scrub	4.7	T	4

(*Continued*)

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

(*Continued*)

Field Name	Area	Slope	Vegetation	Soil pH	Damp	Worm Density
Pound Hill	4.4	2	Arable	4.5	F	5
Gravel Pit	2.9	1	Grassland	3.5	F	1
Farm Wood	0.8	10	Scrub	5.1	T	3

Perhaps the most important thing about analysing your own data properly is getting your dataframe absolutely right. The expectation is that you will have used a spreadsheet such as Excel to enter and edit the data, and that you will have used plots to check for errors. The thing that takes some practice is learning exactly how to put your numbers into the spreadsheet. There are countless ways of doing it wrong, but only one way of doing it right. And this way is *not* the way that most people find intuitively to be the most obvious.

The key thing is this: all the values of the same variable must go in the same column. It does not sound like much, but this is what people tend to get wrong. If you had an experiment with three treatments (control, preheated and prechilled), and four measurements per treatment, it might seem like a good idea to create the spreadsheet like this:

control	preheated	prechilled			
6.1	6.3	7.1			
5.9	6.2	8.2			
5.8	5.8	7.3			
5.4	6.3	6.9			

However, this is not a dataframe, because values of the response variable appear in three different columns, rather than all in the same column. The correct way to enter these data is to have two columns: one for the response variable and one for the levels of the experimental factor called Treatment (control, preheated and prechilled). Here are the same data, entered correctly as a dataframe:

Response	Treatment			
6.1	Control			
5.9	Control			
5.8	Control			
5.4	Control			
6.3	Preheated			
6.2	Preheated			
5.8	Preheated			
6.3	Preheated			
7.1	Prechilled			
8.2	Prechilled			
7.3	Prechilled			
6.9	Prechilled			

A good way to practice this layout is to use the Excel function called PivotTable (found under Data on the main menu bar) on your own data: it requires your spreadsheet to be in the form of a dataframe, with each of the variables in its own column.

Once you have made your dataframe in a spreadsheet 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 the spreadsheet as tab-delimited text files. In Excel, for instance, you click on 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 as explained in Chapter 3.

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), such as Field Name, Soil pH or Worm Density (above), or between any of the words within the same factor level (as in many of the field names). These should be replaced by dots '.' before the dataframe is saved from the spreadsheet. Also, it is good idea to remove any apostrophes, as these can sometimes cause problems because there is more than one ASCII code for quotation marks. Now the dataframe can be read into R. Think of a name for the dataframe (say, 'worms' in this case) and then allocate the data from the file to the dataframe name using the gets arrow `<-` like this:

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

Once the file has been imported to R we often want to do four things:

- use `attach` to make the variables accessible by name within the R session;
- use `names` to get a list of the variable names;
- use `head` to look at the first few rows of the data;
- use `tail` to look at the last few rows of the data.

Typically, the commands are issued in sequence, whenever a new dataframe is imported from file (but see p. 113 for superior alternatives to `attach`):

```
attach(worms)
names(worms)
```

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

```
head(worms)
```

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

```
tail(worms)
```

```
       Field.Name Area Slope Vegetation Soil.pH  Damp Worm.density
15     Pond.Field  4.1     0     Meadow     5.0  TRUE            6
16   Water.Meadow  3.9     0     Meadow     4.9  TRUE            8
17      Cheapside  2.2     8      Scrub     4.7  TRUE            4
18     Pound.Hill  4.4     2     Arable     4.5 FALSE            5
19     Gravel.Pit  2.9     1  Grassland     3.5 FALSE            1
20      Farm.Wood  0.8    10      Scrub     5.1  TRUE            3
```

To see the contents of the whole dataframe, just type its name:

```
worms
```

```
          Field.Name Area Slope Vegetation Soil.pH  Damp Worm.density
1         Nashs.Field  3.6    11  Grassland     4.1 FALSE            4
2      Silwood.Bottom  5.1     2     Arable     5.2 FALSE            7
3       Nursery.Field  2.8     3  Grassland     4.3 FALSE            2
4         Rush.Meadow  2.4     5     Meadow     4.9  TRUE            5
5     Gunness.Thicket  3.8     0      Scrub     4.2 FALSE            6
6            Oak.Mead  3.1     2  Grassland     3.9 FALSE            2
7        Church.Field  3.5     3  Grassland     4.2 FALSE            3
8             Ashurst  2.1     0     Arable     4.8 FALSE            4
9         The.Orchard  1.9     0    Orchard     5.7 FALSE            9
10      Rookery.Slope  1.5     4  Grassland     5.0  TRUE            7
11        Garden.Wood  2.9    10      Scrub     5.2 FALSE            8
12       North.Gravel  3.3     1  Grassland     4.1 FALSE            1
13       South.Gravel  3.7     2  Grassland     4.0 FALSE            2
14 Observatory.Ridge  1.8     6  Grassland     3.8 FALSE            0
15         Pond.Field  4.1     0     Meadow     5.0  TRUE            6
16       Water.Meadow  3.9     0     Meadow     4.9  TRUE            8
17          Cheapside  2.2     8      Scrub     4.7  TRUE            4
18         Pound.Hill  4.4     2     Arable     4.5 FALSE            5
19         Gravel.Pit  2.9     1  Grassland     3.5 FALSE            1
20          Farm.Wood  0.8    10      Scrub     5.1  TRUE            3
```

Notice that R has expanded our abbreviated T and F into `TRUE` and `FALSE`. The object called `worms` now has all the attributes of a dataframe. For example, you can summarize it, using `summary`:

```
summary(worms)
```

```
      Field.Name      Area           Slope           Vegetation
Ashurst     : 1   Min.   :0.800   Min.   : 0.00   Arable   :3
Cheapside   : 1   1st Qu.:2.175   1st Qu.: 0.75   Grassland:9
Church.Field: 1   Median :3.000   Median : 2.00   Meadow   :3
Farm.Wood   : 1   Mean   :2.990   Mean   : 3.50   Orchard  :1
Garden.Wood : 1   3rd Qu.:3.725   3rd Qu.: 5.25   Scrub    :4
Gravel.Pit  : 1   Max.   :5.100   Max.   :11.00
(Other)     :14

 Soil.pH         Damp           Worm.density
Min.   :3.500   Mode :logical   Min.   :0.00
1st Qu.:4.100   FALSE:14        1st Qu.:2.00
Median :4.600   TRUE :6         Median :4.00
Mean   :4.555   NA's :0         Mean   :4.35
3rd Qu.:5.000                   3rd Qu.:6.25
Max.   :5.700                   Max.   :9.00
```

Values of continuous variables are summarized under six headings: one parametric (the arithmetic mean) and five non-parametric (maximum, minimum, median, 25th percentile or first quartile, and 75th percentile or third quartile). Tukey's famous five-number function (`fivenum`; see p. 42) is slightly different, with hinges

rather than first and third quartiles. Levels of categorical variables are counted. Note that the field names are not listed in full because they are unique to each row; six of them are named, then R says 'plus 14 others' `(Other) :14`.

The two functions `by` and `aggregate` allow summary of the dataframe on the basis of factor levels. For instance, it might be interesting to know the means of the numeric variables for each vegetation type. The function for this is `by`:

```
by(worms,Vegetation,mean)
```

```
Vegetation: Arable
Field.Name         Area        Slope   Vegetation      Soil.pH         Damp Worm.density
        NA     3.866667     1.333333           NA     4.833333     0.000000     5.333333
------------------------------------------------------------------------------------------
Vegetation: Grassland
Field.Name         Area        Slope   Vegetation      Soil.pH         Damp Worm.density
        NA    2.9111111    3.6666667           NA    4.1000000    0.1111111    2.4444444
------------------------------------------------------------------------------------------
Vegetation: Meadow
Field.Name         Area        Slope   Vegetation      Soil.pH         Damp Worm.density
        NA     3.466667     1.666667           NA     4.933333     1.000000     6.333333
------------------------------------------------------------------------------------------
Vegetation: Orchard
Field.Name         Area        Slope   Vegetation      Soil.pH         Damp Worm.density
        NA          1.9          0.0           NA          5.7          0.0          9.0
------------------------------------------------------------------------------------------
Vegetation: Scrub
Field.Name         Area        Slope   Vegetation      Soil.pH         Damp Worm.density
        NA        2.425        7.000           NA        4.800        0.500        5.250
```

Notice that the logical variable `Damp` has been coerced to numeric (`TRUE` = 1, `FALSE` = 0) and then averaged. Warning messages are printed for the non-numeric variables to which the function `mean` is not applicable (e.g. the factor levels for `Field.name` and `Vegetation`), but this is a useful and quick overview of the effects of the five types of vegetation.

You can also fit models using `by`: here is worm density as a function of soil pH for each vegetation type:

```
by(worms, Vegetation, function(x) lm(Worm.density ~ Soil.pH, data=x))
```

```
Vegetation: Arable
Call:
lm(formula = Worm.density ~ Soil.pH, data = x)

Coefficients:
(Intercept)      Soil.pH
     -9.689        3.108

------------------------------------------------------------------------------
---------------
etc. for each level of vegetation in alphabetical order
------------------------------------------------------------------------------
---------------
Vegetation: Scrub

Call:
lm(formula = Worm.density ~ Soil.pH, data = x)
```

```
Coefficients:
(Intercept)     Soil.pH
   4.4758        0.1613
```

4.1 Subscripts and indices

The key thing about working effectively with dataframes is to become completely at ease with using subscripts (or indices, as some people call them). In R, subscripts appear in square brackets [ ]. A dataframe is a two-dimensional object, comprising rows and columns. The rows are referred to by the first (left-hand) subscript, the columns by the second (right-hand) subscript. Thus

```
worms[3,5]
```

```
[1]  4.3
```

is the value in row 3 of `Soil.pH` (the variable in column 5). To extract a range of values (say the 14th to 19th rows) from worm density (the variable in the seventh column) we use the colon operator `:` to generate a series of subscripts (14, 15, 16, 17, 18 and 19):

```
worms[14:19,7]
```

```
[1]  0  6  8  4  5  1
```

To extract a group of rows and a group of columns, you need to generate a series of subscripts for both the row and column subscripts. Suppose we want `Area` and `Slope` (columns 2 and 3) from rows 1 to 5:

```
worms[1:5,2:3]
```

```
       Area    Slope
1       3.6       11
2       5.1        2
3       2.8        3
4       2.4        5
5       3.8        0
```

This next point is very important, and is hard to grasp without practice. To select *all* the entries in a *row* the syntax is 'number comma blank'. Similarly, to select all the entries in a *column* the syntax is 'blank comma number'. Thus, to select all the columns in row 3 we type

```
worms[3,]
```

```
     Field.Name Area Slope Vegetation Soil.pH  Damp Worm.density
3 Nursery.Field  2.8     3  Grassland     4.3 FALSE            2
```

whereas to select all the rows in column 3 we need

```
worms[,3]
```

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

This is a key feature of the R language, and one that causes problems for beginners. Note that these two apparently similar commands create *objects of different classes*:

```
class(worms[3,])
```

```
[1]  "data.frame"
```

```
class(worms[,3])
```

```
[1]  "integer"
```

You can create sets of rows or columns. For instance, to extract all the rows for `Field.Name` and `Soil.pH` (columns 1 and 5) use the concatenate function, `c`, to make a vector of the required column numbers `c(1,5)`:

```
worms[,c(1,5)]
```

```
           Field.Name Soil.pH
1         Nashs.Field     4.1
2      Silwood.Bottom     5.2
3       Nursery.Field     4.3
4         Rush.Meadow     4.9
5     Gunness.Thicket     4.2
6             Oak.Mead     3.9
7        Church.Field     4.2
8             Ashurst     4.8
9         The.Orchard     5.7
10      Rookery.Slope     5.0
11        Garden.Wood     5.2
12       North.Gravel     4.1
13       South.Gravel     4.0
14 Observatory.Ridge     3.8
15         Pond.Field     5.0
16       Water.Meadow     4.9
17          Cheapside     4.7
18         Pound.Hill     4.5
19         Gravel.Pit     3.5
20          Farm.Wood     5.1
```

The commands for selecting rows and columns from the dataframe are summarized in Table 4.1.

4.2 Selecting rows from the dataframe at random

In bootstrapping or cross-validation we might want to select certain rows from the dataframe at random. We use the `sample` function to do this: the default `replace = FALSE` performs shuffling (each row is selected once and only once), while the option `replace = TRUE` (sampling with replacement) allows for multiple copies of certain rows and the omission of others. Here we use the default `replace = F` to select a unique 8 of the 20 rows at random:

```
worms[sample(1:20,8),]
```

```
       Field.Name Area Slope Vegetation Soil.pH  Damp Worm.density
7    Church.Field  3.5     3  Grassland     4.2 FALSE             3
3   Nursery.Field  2.8     3  Grassland     4.3 FALSE             2
19     Gravel.Pit  2.9     1  Grassland     3.5 FALSE             1
4     Rush.Meadow  2.4     5     Meadow     4.9  TRUE             5
18     Pound.Hill  4.4     2     Arable     4.5 FALSE             5
```

```
12  North.Gravel  3.3     1  Grassland     4.1 FALSE            1
1    Nashs.Field  3.6    11  Grassland     4.1 FALSE            4
15    Pond.Field  4.1     0     Meadow     5.0  TRUE            6
```

Table 4.1. Selecting parts of a dataframe called `data`. Suppose that `n` is one of the row numbers in your dataframe that you want to select or remove, and `m` is one of the columns. Note that the syntax `[n,]` selects all of the columns, while `[,m]` selects all of the rows.

command	meaning
`data[n,]`	select all of the columns from row *n* of the dataframe
`data[-n,]`	drop the whole of row *n* from the dataframe
`data[1:n,]`	select all of the columns from rows 1 to *n* of the dataframe
`data[-(1:n),]`	drop all of the columns from rows 1 to *n* of the dataframe
`data[c(i,j,k),]`	select all of the columns from rows *i*, *j*, and *k* of the dataframe
`data[x > y,]`	use a logical test ($x > y$) to select all columns from certain rows
`data[,m]`	select all of the rows from column *m* of the dataframe
`data[,-m]`	drop the whole of column *m* from the dataframe
`data[,1:m]`	select all of the rows from columns 1 to *m* of the dataframe
`data[,-(1:m)]`	drop all of the rows from columns 1 to *m* of the dataframe
`data[,c(i,j,k)]`	select all of the rows from columns *i*, *j*, and *k* of the dataframe
`data[,x > y]`	use a logical test ($x > y$) to select all rows from certain columns
`data[,c(1:m,i,j,k)]`	add duplicate copies of columns *i*, *j*, and *k* to the dataframe
`data[x > y,a != b]`	extract certain rows ($x > y$) and certain columns (a != b)
`data[c(1:n,i,j,k),]`	add duplicate copies of rows *i*, *j*, and *k* to the dataframe

Note that the row numbers are in random sequence (not sorted), so that if you want a sorted random sample you will need to order the dataframe after the randomization.

4.3 Sorting dataframes

It is common to want to sort a dataframe by rows, but rare to want to sort by columns. Because we are sorting by rows (the first subscript) we specify the order of the row subscripts *before* the comma. Thus, to sort the dataframe on the basis of values in one of the columns (say, `Slope`), we write

```
worms[order(Slope),]
```

```
           Field.Name Area Slope Vegetation Soil.pH  Damp Worm.density
5    Gunness.Thicket  3.8     0      Scrub     4.2 FALSE            6
8            Ashurst  2.1     0     Arable     4.8 FALSE            4
9         The.Orchard  1.9     0    Orchard     5.7 FALSE            9
15         Pond.Field  4.1     0     Meadow     5.0  TRUE            6
16       Water.Meadow  3.9     0     Meadow     4.9  TRUE            8
12       North.Gravel  3.3     1  Grassland     4.1 FALSE            1
19         Gravel.Pit  2.9     1  Grassland     3.5 FALSE            1
2     Silwood.Bottom  5.1     2     Arable     5.2 FALSE            7
6            Oak.Mead  3.1     2  Grassland     3.9 FALSE            2
13       South.Gravel  3.7     2  Grassland     4.0 FALSE            2
18         Pound.Hill  4.4     2     Arable     4.5 FALSE            5
3       Nursery.Field  2.8     3  Grassland     4.3 FALSE            2
7        Church.Field  3.5     3  Grassland     4.2 FALSE            3
```

```
10      Rookery.Slope  1.5     4  Grassland     5.0  TRUE            7
4         Rush.Meadow  2.4     5     Meadow     4.9  TRUE            5
14 Observatory.Ridge  1.8     6  Grassland     3.8 FALSE            0
17          Cheapside  2.2     8      Scrub     4.7  TRUE            4
11        Garden.Wood  2.9    10      Scrub     5.2 FALSE            8
20          Farm.Wood  0.8    10      Scrub     5.1  TRUE            3
1         Nashs.Field  3.6    11  Grassland     4.1 FALSE            4
```

There are some points to notice here. Because we wanted the sorting to apply to all the columns, the column subscript (after the comma) is blank: `[order(Slope),]`. The original row numbers are retained in the leftmost column. Where there are ties for the sorting variable (e.g. there are five ties for `Slope` = 0) then the rows are in their original order. If you want the dataframe in reverse order (ascending order) then use the `rev` function outside the `order` function like this:

```
worms[rev(order(Slope)),]
```

```
           Field.Name Area Slope Vegetation Soil.pH  Damp Worm.density
1         Nashs.Field  3.6    11  Grassland     4.1 FALSE            4
20          Farm.Wood  0.8    10      Scrub     5.1  TRUE            3
11        Garden.Wood  2.9    10      Scrub     5.2 FALSE            8
17          Cheapside  2.2     8      Scrub     4.7  TRUE            4
14 Observatory.Ridge  1.8     6  Grassland     3.8 FALSE            0
4         Rush.Meadow  2.4     5     Meadow     4.9  TRUE            5
10      Rookery.Slope  1.5     4  Grassland     5.0  TRUE            7
7        Church.Field  3.5     3  Grassland     4.2 FALSE            3
3       Nursery.Field  2.8     3  Grassland     4.3 FALSE            2
18          Pound.Hill  4.4     2     Arable     4.5 FALSE            5
13       South.Gravel  3.7     2  Grassland     4.0 FALSE            2
6             Oak.Mead  3.1     2  Grassland     3.9 FALSE            2
2      Silwood.Bottom  5.1     2     Arable     5.2 FALSE            7
19         Gravel.Pit  2.9     1  Grassland     3.5 FALSE            1
12       North.Gravel  3.3     1  Grassland     4.1 FALSE            1
16       Water.Meadow  3.9     0     Meadow     4.9  TRUE            8
15         Pond.Field  4.1     0     Meadow     5.0  TRUE            6
9         The.Orchard  1.9     0    Orchard     5.7 FALSE            9
8               Ashurst  2.1     0     Arable     4.8 FALSE            4
5     Gunness.Thicket  3.8     0      Scrub     4.2 FALSE            6
```

Notice now that when there are ties (e.g. `Slope` = 0), the original rows are also in reverse order.

More complicated sorting operations might involve two or more variables. This is achieved very simply by separating a series of variable names by commas within the `order` function. R will sort on the basis of the left-hand variable, with ties being broken by the second variable, and so on. Suppose that we want to order the rows of the database on worm density within each vegetation type:

```
worms[order(Vegetation,Worm.density),]
```

```
           Field.Name Area Slope Vegetation Soil.pH  Damp Worm.density
8               Ashurst  2.1     0     Arable     4.8 FALSE            4
18          Pound.Hill  4.4     2     Arable     4.5 FALSE            5
2      Silwood.Bottom  5.1     2     Arable     5.2 FALSE            7
14 Observatory.Ridge  1.8     6  Grassland     3.8 FALSE            0
```

```
12       North.Gravel  3.3      1  Grassland      4.1 FALSE             1
19         Gravel.Pit  2.9      1  Grassland      3.5 FALSE             1
3       Nursery.Field  2.8      3  Grassland      4.3 FALSE             2
6            Oak.Mead  3.1      2  Grassland      3.9 FALSE             2
13       South.Gravel  3.7      2  Grassland      4.0 FALSE             2
7        Church.Field  3.5      3  Grassland      4.2 FALSE             3
1         Nashs.Field  3.6     11  Grassland      4.1 FALSE             4
10      Rookery.Slope  1.5      4  Grassland      5.0  TRUE             7
4         Rush.Meadow  2.4      5     Meadow      4.9  TRUE             5
15         Pond.Field  4.1      0     Meadow      5.0  TRUE             6
16       Water.Meadow  3.9      0     Meadow      4.9  TRUE             8
9         The.Orchard  1.9      0    Orchard      5.7 FALSE             9
20          Farm.Wood  0.8     10      Scrub      5.1  TRUE             3
17          Cheapside  2.2      8      Scrub      4.7  TRUE             4
5     Gunness.Thicket  3.8      0      Scrub      4.2 FALSE             6
11        Garden.Wood  2.9     10      Scrub      5.2 FALSE             8
```

Notice that as with single-condition sorts, when there are ties (as in grassland with worm density = 2), the rows are in their original sequence (here, 3, 6, 13). We might want to override this by specifying a third sorting condition (e.g. soil pH):

```
worms[order(Vegetation,Worm.density,Soil.pH),]
```

```
           Field.Name Area Slope Vegetation Soil.pH  Damp Worm.density
8             Ashurst  2.1     0     Arable     4.8 FALSE             4
18         Pound.Hill  4.4     2     Arable     4.5 FALSE             5
2     Silwood.Bottom  5.1     2     Arable     5.2 FALSE             7
14 Observatory.Ridge  1.8     6  Grassland     3.8 FALSE             0
19         Gravel.Pit  2.9     1  Grassland     3.5 FALSE             1
12       North.Gravel  3.3     1  Grassland     4.1 FALSE             1
6             Oak.Mead  3.1     2  Grassland     3.9 FALSE             2
13       South.Gravel  3.7     2  Grassland     4.0 FALSE             2
3       Nursery.Field  2.8     3  Grassland     4.3 FALSE             2
7        Church.Field  3.5     3  Grassland     4.2 FALSE             3
1         Nashs.Field  3.6    11  Grassland     4.1 FALSE             4
10      Rookery.Slope  1.5     4  Grassland     5.0  TRUE             7
4         Rush.Meadow  2.4     5     Meadow     4.9  TRUE             5
15         Pond.Field  4.1     0     Meadow     5.0  TRUE             6
16       Water.Meadow  3.9     0     Meadow     4.9  TRUE             8
9         The.Orchard  1.9     0    Orchard     5.7 FALSE             9
20          Farm.Wood  0.8    10      Scrub     5.1  TRUE             3
17          Cheapside  2.2     8      Scrub     4.7  TRUE             4
5     Gunness.Thicket  3.8     0      Scrub     4.2 FALSE             6
11        Garden.Wood  2.9    10      Scrub     5.2 FALSE             8
```

The rule is this: if in doubt, sort using more variables than you think you need. That way you can be absolutely certain that the rows are in the order you expect them to be in. This is exceptionally important when you begin to make assumptions about the variables associated with a particular value of the response variable on the basis of its row number.

Perhaps you want only certain columns in the sorted dataframe? Suppose we want vegetation, worm.density, soil pH and slope, and we want them in that order from left to right. We specify the column numbers in the sequence we want them to appear as a vector: `c(4,7,5,3)`:

```
worms[order(Vegetation,Worm.density),c(4,7,5,3)]
```

```
   Vegetation Worm.density Soil.pH Slope
8      Arable            4     4.8     0
18     Arable            5     4.5     2
2      Arable            7     5.2     2
14  Grassland            0     3.8     6
12  Grassland            1     4.1     1
19  Grassland            1     3.5     1
3   Grassland            2     4.3     3
6   Grassland            2     3.9     2
13  Grassland            2     4.0     2
7   Grassland            3     4.2     3
1   Grassland            4     4.1    11
10  Grassland            7     5.0     4
4      Meadow            5     4.9     5
15     Meadow            6     5.0     0
16     Meadow            8     4.9     0
9     Orchard            9     5.7     0
20      Scrub            3     5.1    10
17      Scrub            4     4.7     8
5       Scrub            6     4.2     0
11      Scrub            8     5.2    10
```

You can select the columns on the basis of their variables names, but this is more fiddly to type, because you need to put the variable names in quotes like this:

```
worms[order(Vegetation,Worm.density),
              c("Vegetation", "Worm.density", "Soil.pH", "Slope")]
```

4.4 Using logical conditions to select rows from the dataframe

A very common operation is selecting certain rows from the dataframe on the basis of values in one or more of the variables (the columns of the dataframe). Suppose we want to restrict the data to cases from damp fields. We want all the columns, so the syntax for the subscripts is ['which rows', blank]:

```
worms[Damp == T,]
```

```
      Field.Name Area Slope Vegetation Soil.pH Damp Worm.density
4    Rush.Meadow  2.4     5     Meadow     4.9 TRUE            5
10 Rookery.Slope  1.5     4  Grassland     5.0 TRUE            7
15    Pond.Field  4.1     0     Meadow     5.0 TRUE            6
16  Water.Meadow  3.9     0     Meadow     4.9 TRUE            8
17     Cheapside  2.2     8      Scrub     4.7 TRUE            4
20     Farm.Wood  0.8    10      Scrub     5.1 TRUE            3
```

Note that because Damp is a logical variable (with just two potential values, TRUE or FALSE) we can refer to true or false in abbreviated form, T or F. Also notice that the T in this case is not enclosed in quotes: the T means true, not the character string "T". The other important point is that the symbol for the logical condition is == (two successive equals signs with no gap between them; see p. 26).

The logic for the selection of rows can refer to values (and functions of values) in more than one column. Suppose that we wanted the data from the fields where worm density was higher than the median (>median(Worm.density)) and soil pH was less than 5.2. In R, the logical operator for AND is the & ('ampersand') symbol:

```
worms[Worm.density > median(Worm.density) & Soil.pH < 5.2,]
```

```
         Field.Name Area Slope Vegetation Soil.pH  Damp Worm.density
4       Rush.Meadow  2.4     5     Meadow     4.9  TRUE            5
5  Gunness.Thicket  3.8     0      Scrub     4.2 FALSE            6
10   Rookery.Slope  1.5     4  Grassland     5.0  TRUE            7
15      Pond.Field  4.1     0     Meadow     5.0  TRUE            6
16    Water.Meadow  3.9     0     Meadow     4.9  TRUE            8
18      Pound.Hill  4.4     2     Arable     4.5 FALSE            5
```

Suppose that we want to extract all the columns that contain numbers (rather than characters or logical variables) from the dataframe. The function is.numeric can be applied across all the columns of worms using sapply to create the appropriate subscripts like this:

```
worms[,sapply(worms,is.numeric)]
```

```
   Area Slope Soil.pH Worm.density
1   3.6    11     4.1            4
2   5.1     2     5.2            7
3   2.8     3     4.3            2
4   2.4     5     4.9            5
5   3.8     0     4.2            6
6   3.1     2     3.9            2
7   3.5     3     4.2            3
8   2.1     0     4.8            4
9   1.9     0     5.7            9
10  1.5     4     5.0            7
11  2.9    10     5.2            8
12  3.3     1     4.1            1
13  3.7     2     4.0            2
14  1.8     6     3.8            0
15  4.1     0     5.0            6
16  3.9     0     4.9            8
17  2.2     8     4.7            4
18  4.4     2     4.5            5
19  2.9     1     3.5            1
20  0.8    10     5.1            3
```

We might want to extract the columns that were factors:

```
worms[,sapply(worms,is.factor)]
```

```
          Field.Name Vegetation
1        Nashs.Field  Grassland
2     Silwood.Bottom     Arable
3      Nursery.Field  Grassland
4        Rush.Meadow     Meadow
5    Gunness.Thicket      Scrub
6           Oak.Mead  Grassland
7       Church.Field  Grassland
8             Ashurst     Arable
9        The.Orchard    Orchard
10     Rookery.Slope  Grassland
11       Garden.Wood      Scrub
12      North.Gravel  Grassland
13      South.Gravel  Grassland
14 Observatory.Ridge  Grassland
15        Pond.Field     Meadow
16      Water.Meadow     Meadow
17         Cheapside      Scrub
18        Pound.Hill     Arable
19        Gravel.Pit  Grassland
20         Farm.Wood      Scrub
```

Because worms is a dataframe, the characters have all been coerced to factors, so `worms[,sapply(worms,is.character)]` produces the answer `NULL`.

To drop a row or rows from the dataframe, use **negative subscripts**. Thus to drop the middle 10 rows (i.e. row numbers 6 to 15 inclusive) do this:

```
worms[-(6:15),]
```

```
        Field.Name Area Slope Vegetation Soil.pH  Damp Worm.density
1      Nashs.Field  3.6    11  Grassland     4.1 FALSE            4
2   Silwood.Bottom  5.1     2     Arable     5.2 FALSE            7
3    Nursery.Field  2.8     3  Grassland     4.3 FALSE            2
4      Rush.Meadow  2.4     5     Meadow     4.9  TRUE            5
5  Gunness.Thicket  3.8     0      Scrub     4.2 FALSE            6
16    Water.Meadow  3.9     0     Meadow     4.9  TRUE            8
17       Cheapside  2.2     8      Scrub     4.7  TRUE            4
18      Pound.Hill  4.4     2     Arable     4.5 FALSE            5
19      Gravel.Pit  2.9     1  Grassland     3.5 FALSE            1
20       Farm.Wood  0.8    10      Scrub     5.1  TRUE            3
```

Here are all the rows that are not grasslands (recall that the logical symbol `!` means NOT):

```
worms[!(Vegetation=="Grassland"),]
```

```
        Field.Name Area Slope Vegetation Soil.pH  Damp Worm.density
2   Silwood.Bottom  5.1     2     Arable     5.2 FALSE            7
4      Rush.Meadow  2.4     5     Meadow     4.9  TRUE            5
5  Gunness.Thicket  3.8     0      Scrub     4.2 FALSE            6
8          Ashurst  2.1     0     Arable     4.8 FALSE            4
9      The.Orchard  1.9     0    Orchard     5.7 FALSE            9
11     Garden.Wood  2.9    10      Scrub     5.2 FALSE            8
```

```
15      Pond.Field  4.1     0     Meadow     5.0  TRUE            6
16    Water.Meadow  3.9     0     Meadow     4.9  TRUE            8
17       Cheapside  2.2     8      Scrub     4.7  TRUE            4
18      Pound.Hill  4.4     2     Arable     4.5 FALSE            5
20       Farm.Wood  0.8    10      Scrub     5.1  TRUE            3
```

If you want to use minus signs rather than logical NOT to drop rows from the dataframe, the expression you use must evaluate to numbers. The `which` function is useful for this. Let us use this technique to drop the non-damp fields:

```
worms[-which(Damp==F),]
```

```
      Field.Name Area Slope Vegetation Soil.pH Damp Worm.density
4    Rush.Meadow  2.4     5     Meadow     4.9 TRUE            5
10 Rookery.Slope  1.5     4  Grassland     5.0 TRUE            7
15    Pond.Field  4.1     0     Meadow     5.0 TRUE            6
16  Water.Meadow  3.9     0     Meadow     4.9 TRUE            8
17     Cheapside  2.2     8      Scrub     4.7 TRUE            4
20     Farm.Wood  0.8    10      Scrub     5.1 TRUE            3
```

which achieves the same end as the more elegant

```
worms[!Damp==F,]
```

or, even simpler,

```
worms[Damp==T,]
```

4.5 Omitting rows containing missing values, `NA`

In statistical modelling it is often useful to have a dataframe that contains no missing values in the response or explanatory variables. You can create a shorter dataframe using the `na.omit` function. Here is a sister dataframe of `worms` in which certain values are `NA`:

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

```
          Field.Name Area Slope Vegetation Soil.pH  Damp Worm.density
1        Nashs.Field  3.6    11  Grassland     4.1 FALSE            4
2     Silwood.Bottom  5.1    NA     Arable     5.2 FALSE            7
3      Nursery.Field  2.8     3  Grassland     4.3 FALSE            2
4        Rush.Meadow  2.4     5     Meadow     4.9  TRUE            5
5    Gunness.Thicket  3.8     0      Scrub     4.2 FALSE            6
6           Oak.Mead  3.1     2  Grassland     3.9 FALSE            2
7       Church.Field  3.5     3  Grassland      NA    NA           NA
8            Ashurst  2.1     0     Arable     4.8 FALSE            4
9        The.Orchard  1.9     0    Orchard     5.7 FALSE            9
10     Rookery.Slope  1.5     4  Grassland     5.0  TRUE            7
11       Garden.Wood  2.9    10      Scrub     5.2 FALSE            8
12      North.Gravel  3.3     1  Grassland     4.1 FALSE            1
13      South.Gravel  3.7     2  Grassland     4.0 FALSE            2
```

```
14 Observatory.Ridge  1.8     6  Grassland     3.8 FALSE            0
15         Pond.Field  4.1     0     Meadow     5.0  TRUE            6
16       Water.Meadow  3.9     0     Meadow     4.9  TRUE            8
17          Cheapside  2.2     8      Scrub     4.7  TRUE            4
18         Pound.Hill  4.4     2     Arable     4.5 FALSE            5
19         Gravel.Pit   NA     1  Grassland     3.5 FALSE            1
20          Farm.Wood  0.8    10      Scrub     5.1  TRUE            3
```

By inspection we can see that we should like to leave out row 2 (one missing value), row 7 (three missing values) and row 19 (one missing value). This could not be simpler:

```
na.omit(data)
```

```
           Field.Name Area Slope Vegetation Soil.pH  Damp Worm.density
1         Nashs.Field  3.6    11  Grassland     4.1 FALSE            4
3       Nursery.Field  2.8     3  Grassland     4.3 FALSE            2
4         Rush.Meadow  2.4     5     Meadow     4.9  TRUE            5
5     Gunness.Thicket  3.8     0      Scrub     4.2 FALSE            6
6            Oak.Mead  3.1     2  Grassland     3.9 FALSE            2
8             Ashurst  2.1     0     Arable     4.8 FALSE            4
9         The.Orchard  1.9     0    Orchard     5.7 FALSE            9
10      Rookery.Slope  1.5     4  Grassland     5.0  TRUE            7
11        Garden.Wood  2.9    10      Scrub     5.2 FALSE            8
12       North.Gravel  3.3     1  Grassland     4.1 FALSE            1
13       South.Gravel  3.7     2  Grassland     4.0 FALSE            2
14 Observatory.Ridge  1.8     6  Grassland     3.8 FALSE            0
15         Pond.Field  4.1     0     Meadow     5.0  TRUE            6
16       Water.Meadow  3.9     0     Meadow     4.9  TRUE            8
17          Cheapside  2.2     8      Scrub     4.7  TRUE            4
18         Pound.Hill  4.4     2     Arable     4.5 FALSE            5
20          Farm.Wood  0.8    10      Scrub     5.1  TRUE            3
```

and you see that rows 2, 7 and 19 have been omitted in creating the new dataframe. Alternatively, you can use the `na.exclude` function. This differs from `na.omit` only in the class of the `na.action` attribute of the result, which gives different behaviour in functions making use of `naresid` and `napredict`: when `na.exclude` is used the residuals and predictions are padded to the correct length by inserting `NA`s for cases omitted by `na.exclude` (in this example they would be of length 20, whereas `na.omit` would give residuals and predictions of length 17).

```
new.frame <- na.exclude(data)
```

The function to test for the presence of missing values across a dataframe is `complete.cases`:

```
complete.cases(data)
```

```
 [1]   TRUE FALSE  TRUE  TRUE  TRUE  TRUE FALSE  TRUE  TRUE  TRUE  TRUE
[12]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE  TRUE
```

You could use this as a less efficient analogue of `na.omit(data)`, but why would you?

```
data[complete.cases(data),]
```

It is well worth checking the individual variables separately, because it is possible that one or more variables contribute most of the missing values, and it may be preferable to remove these variables from the modelling rather than lose the valuable information about the other explanatory variables associated with these cases. Use `summary` to count the missing values for each variable in the dataframe, or use `apply` with the function `is.na` to sum the missing values in each variable:

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

```
Field.Name   Area  Slope  Vegetation  Soil.pH  Damp  Worm.density
         0      1      1           0        1     1             1
```

You can see that in this case no single variable contributed more missing values than any other.

4.5.1 Replacing NAs with zeros

You would need to think carefully before doing this, but there might be circumstances when you wanted to replace the missing values `NA` by zero (or by some other missing-value indicator). Continuing the missing-worms example, above, where the dataframe called `data` contained five missing values, this is how to replace all the `NA`s by zeros:

```
data[is.na(data)]<-0
```

4.6 Using `order` and `!duplicated` to eliminate pseudoreplication

In this rather more complicated example, you are asked to extract a single record for each vegetation type, and that record is to be the case within each vegetation type that has the greatest worm density. There are two steps to this: first order all of the rows in a new dataframe using `rev(order(Worm.density))`, then select the subset of these rows which is not duplicated (`!duplicated`) within each vegetation type in the new dataframe (using `new$Vegetation`) :

```
new <- worms[rev(order(Worm.density)),]
new[!duplicated(new$Vegetation),]
```

```
       Field.Name Area Slope Vegetation Soil.pH  Damp Worm.density
9     The.Orchard  1.9     0    Orchard     5.7 FALSE            9
16   Water.Meadow  3.9     0     Meadow     4.9  TRUE            8
11    Garden.Wood  2.9    10      Scrub     5.2 FALSE            8
10  Rookery.Slope  1.5     4  Grassland     5.0  TRUE            7
2  Silwood.Bottom  5.1     2     Arable     5.2 FALSE            7
```

4.7 Complex ordering with mixed directions

Sometimes there are multiple sorting variables, but the variables have to be sorted in opposing directions. In this example, the task is to order the database first by vegetation type in alphabetical order (the default) and then within each vegetation type to sort by worm density in decreasing order (highest densities first). The trick here is to use `order` (rather than `rev(order)`) but to put a minus sign in front of `Worm.density` like this:

```
worms[order(Vegetation,-Worm.density),]
```

```
           Field.Name Area Slope Vegetation Soil.pH  Damp Worm.density
2      Silwood.Bottom  5.1     2     Arable     5.2 FALSE             7
18         Pound.Hill  4.4     2     Arable     4.5 FALSE             5
8             Ashurst  2.1     0     Arable     4.8 FALSE             4
10      Rookery.Slope  1.5     4  Grassland     5.0  TRUE             7
1         Nashs.Field  3.6    11  Grassland     4.1 FALSE             4
7        Church.Field  3.5     3  Grassland     4.2 FALSE             3
3       Nursery.Field  2.8     3  Grassland     4.3 FALSE             2
6            Oak.Mead  3.1     2  Grassland     3.9 FALSE             2
13       South.Gravel  3.7     2  Grassland     4.0 FALSE             2
12       North.Gravel  3.3     1  Grassland     4.1 FALSE             1
19         Gravel.Pit  2.9     1  Grassland     3.5 FALSE             1
14 Observatory.Ridge  1.8     6  Grassland     3.8 FALSE             0
16       Water.Meadow  3.9     0     Meadow     4.9  TRUE             8
15         Pond.Field  4.1     0     Meadow     5.0  TRUE             6
4         Rush.Meadow  2.4     5     Meadow     4.9  TRUE             5
9         The.Orchard  1.9     0    Orchard     5.7 FALSE             9
11        Garden.Wood  2.9    10      Scrub     5.2 FALSE             8
5     Gunness.Thicket  3.8     0      Scrub     4.2 FALSE             6
17          Cheapside  2.2     8      Scrub     4.7  TRUE             4
20          Farm.Wood  0.8    10      Scrub     5.1  TRUE             3
```

Using the minus sign only works when sorting numerical variables. For factor levels you can use the `rank` function to make the levels numeric like this:

```
worms[order(-rank(Vegetation),-Worm.density),]
```

```
           Field.Name Area Slope Vegetation Soil.pH  Damp Worm.density
11        Garden.Wood  2.9    10      Scrub     5.2 FALSE             8
5     Gunness.Thicket  3.8     0      Scrub     4.2 FALSE             6
17          Cheapside  2.2     8      Scrub     4.7  TRUE             4
20          Farm.Wood  0.8    10      Scrub     5.1  TRUE             3
9         The.Orchard  1.9     0    Orchard     5.7 FALSE             9
16       Water.Meadow  3.9     0     Meadow     4.9  TRUE             8
15         Pond.Field  4.1     0     Meadow     5.0  TRUE             6
4         Rush.Meadow  2.4     5     Meadow     4.9  TRUE             5
10      Rookery.Slope  1.5     4  Grassland     5.0  TRUE             7
1         Nashs.Field  3.6    11  Grassland     4.1 FALSE             4
7        Church.Field  3.5     3  Grassland     4.2 FALSE             3
3       Nursery.Field  2.8     3  Grassland     4.3 FALSE             2
6            Oak.Mead  3.1     2  Grassland     3.9 FALSE             2
13       South.Gravel  3.7     2  Grassland     4.0 FALSE             2
12       North.Gravel  3.3     1  Grassland     4.1 FALSE             1
19         Gravel.Pit  2.9     1  Grassland     3.5 FALSE             1
14 Observatory.Ridge  1.8     6  Grassland     3.8 FALSE             0
2      Silwood.Bottom  5.1     2     Arable     5.2 FALSE             7
18         Pound.Hill  4.4     2     Arable     4.5 FALSE             5
8             Ashurst  2.1     0     Arable     4.8 FALSE             4
```

It is less likely that you will want to select *columns* on the basis of logical operations, but it is perfectly possible. Suppose that for some reason you want to select the columns that contain the character 'S' (upper-case S). In R the function for this is `grep`, which returns the subscript (a number or set of numbers) indicating which character strings within a vector of character strings contained an upper-case S. The names of the variables within a dataframe are obtained by the `names` function:

```
names(worms)
```

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

so we want our function `grep` to pick out variables numbers 3 and 5 because they are the only ones containing upper-case S:

```
grep("S",names(worms))
```

```
[1]  3  5
```

Finally, we can use these numbers as subscripts `[,c(3,5)]` to select columns 3 and 5:

```
worms[,grep("S",names(worms))]
```

```
   Slope Soil.pH
1     11     4.1
2      2     5.2
3      3     4.3
4      5     4.9
5      0     4.2
6      2     3.9
7      3     4.2
8      0     4.8
9      0     5.7
10     4     5.0
11    10     5.2
12     1     4.1
13     2     4.0
14     6     3.8
15     0     5.0
16     0     4.9
17     8     4.7
18     2     4.5
19     1     3.5
20    10     5.1
```

4.8 A dataframe with row names instead of row numbers

You can suppress the creation of row numbers and allocate your own unique names to each row by altering the syntax of the `read.table` function. The first column of the worms database contains the names of the fields in which the other variables were measured. Up to now, we have read this column as if it was the first variable (p. 161).

```
worms2 <- read.table("c:\\temp\\worms.txt",header=T,row.names=1)
worms2
```

```
                  Area Slope Vegetation Soil.pH  Damp Worm.density
Nashs.Field        3.6    11  Grassland     4.1 FALSE            4
Silwood.Bottom     5.1     2     Arable     5.2 FALSE            7
Nursery.Field      2.8     3  Grassland     4.3 FALSE            2
Rush.Meadow        2.4     5     Meadow     4.9  TRUE            5
Gunness.Thicket    3.8     0      Scrub     4.2 FALSE            6
Oak.Mead           3.1     2  Grassland     3.9 FALSE            2
Church.Field       3.5     3  Grassland     4.2 FALSE            3
Ashurst            2.1     0     Arable     4.8 FALSE            4
The.Orchard        1.9     0    Orchard     5.7 FALSE            9
Rookery.Slope      1.5     4  Grassland     5.0  TRUE            7
Garden.Wood        2.9    10      Scrub     5.2 FALSE            8
North.Gravel       3.3     1  Grassland     4.1 FALSE            1
South.Gravel       3.7     2  Grassland     4.0 FALSE            2
Observatory.Ridge  1.8     6  Grassland     3.8 FALSE            0
Pond.Field         4.1     0     Meadow     5.0  TRUE            6
Water.Meadow       3.9     0     Meadow     4.9  TRUE            8
Cheapside          2.2     8      Scrub     4.7  TRUE            4
Pound.Hill         4.4     2     Arable     4.5 FALSE            5
Gravel.Pit         2.9     1  Grassland     3.5 FALSE            1
Farm.Wood          0.8    10      Scrub     5.1  TRUE            3
```

Notice that the field names column is not now headed by a variable name, and that the row numbers, as intended, have been suppressed.

4.9 Creating a dataframe from another kind of object

We have seen that the simplest way to create a dataframe in R is to read a table of data from an external file using the `read.table` function. Alternatively, you can create a dataframe by using the `data.frame` function to bind together a number of vectors. Here are three vectors of the same length:

```
x <- runif(10)
y <- letters[1:10]
z <- sample(c(rep(T,5),rep(F,5)))
```

To make them into a dataframe called `new`, just type:

```
new <- data.frame(y,z,x)
new
```

```
   y     z          x
1  a  TRUE 0.72675982
2  b FALSE 0.83847227
```

```
3        c        FALSE        0.61765685
4        d         TRUE        0.78541650
5        e        FALSE        0.51168828
6        f         TRUE        0.53526324
7        g         TRUE        0.05552335
8        h         TRUE        0.78486234
9        i        FALSE        0.68385443
10       j        FALSE        0.89367837
```

Note that the order of the columns is controlled simply by the sequence of the vector names (left to right) specified within the `data.frame` function.

In this next example, we create a table of counts of random integers from a Poisson distribution, then convert the table into a dataframe. First, we make a table object:

```
y <- rpois(1500,1.5)
table(y)
```

```
y
  0       1       2       3       4       5       6       7
344     502     374     199      63      11       5       2
```

Now it is simple to convert this table object into a dataframe with two variables, the count and the frequency, using the `as.data.frame` function:

```
short<-as.data.frame(table(y))
short
```

```
      y    Freq
1     0     344
2     1     502
3     2     374
4     3     199
5     4      63
6     5      11
7     6       5
8     7       2
```

In some cases you might want to expand a dataframe like the one above such that it had a separate row for every distinct count (i.e. 344 rows with $y = 0$, 502 rows with $y = 1$, 374 rows with $y = 2$, and so on). This is very straightforward using subscripts. We need to create a vector of indices containing 344 repeats of 1, 502 repeats of 2 and so on. Note that these repeats are of the row numbers (1, 2, 3, ..., 8), not repeats of the values of `y` (0, 1, 2, ..., 7).

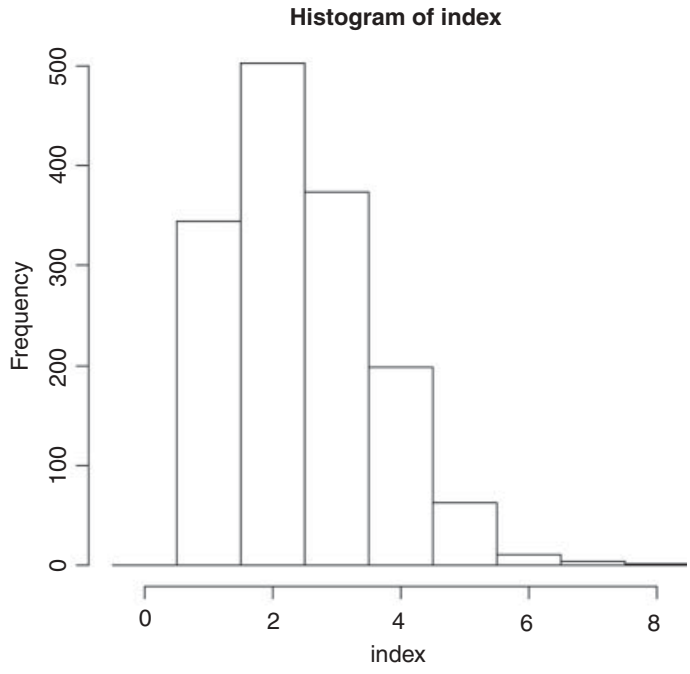
```
index<-rep(1:8,short$Freq)
```

This simple command has produced a vector with the right number of repeats of each of the row numbers

```
length(index)
```

```
[1] 1500
```

```
hist(index,-0.5:8.5)
```



To get the long version of the dataframe, we just use index as the row specifier [index,]:

```
long<-short[index,]
```

Here is a look at the bottom of this long dataframe:

```
tail(long)
```

```
    y Freq
7.1 6    5
7.2 6    5
7.3 6    5
7.4 6    5
8   7    2
8.1 7    2
```

Note the way that R has handled the duplicate row numbers, creating a nested series to indicate the repeats of each of the original row numbers.

A longer-winded alternative might use `lapply` with `rep` to do the same thing:

```
long2 <- as.data.frame(lapply(short,
                 function(x) rep(x, short$Freq)))
tail(long2)
```

```
     y Freq
1495 6    5
1496 6    5
1497 6    5
1498 6    5
```

```
1499 7    2
1500 7    2
```

Note the use of the anonymous function in `lapply` to generate the repeats of each row by the value specified in `Freq`. Before you did anything useful with this longer dataframe, you would probably want to get rid of the redundant column called `Freq`.

4.10 Eliminating duplicate rows from a dataframe

Sometimes a dataframe will contain duplicate rows where all the variables have exactly the same values in two or more rows. Here is a simple example:

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

```
   var1  var2  var3  Var4
1     1     2     3     1
2     1     2     2     1
3     3     2     1     1
4     4     4     2     1
5     3     2     1     1
6     6     1     2     5
7     1     2     3     2
```

Note that row number 5 is an exact duplicate of row number 3. To create a dataframe with all the duplicate rows stripped out, use the `unique` function like this:

```
unique(dups)
```

```
    var1  var2  var3  var4
1      1     2     3     1
2      1     2     2     1
3      3     2     1     1
4      4     4     2     1
6      6     1     2     5
7      1     2     3     2
```

Notice that the row names in the new dataframe are the same as in the original, so that you can spot that row number 5 was removed by the operation of the function `unique`.

To view the rows that are duplicates in a dataframe (if any) use the `duplicated` function to create a vector of `TRUE` and `FALSE` to act as the filter:

```
dups[duplicated(dups),]
```

```
     var1   var2   var3  var4
5       3      2      1     1
```

4.11 Dates in dataframes

There is an introduction to the complexities of using dates and times on pp. 101–113. Here we illustrate a simple example:

```
nums <- read.table("c:\\temp\\sortdata.txt",header=T)
attach(nums)
head(nums)
```

```
     name       date   response treatment
1  albert 25/08/2003 0.05963704         A
2     ann 21/05/2003 1.46555993         A
3    john 12/10/2003 1.59406539         B
4     ian 02/12/2003 2.09505949         A
5 michael 18/10/2003 2.38330748         B
6     ann 02/07/2003 2.86983693         B
```

The idea is to order the rows by date. The ordering is to be applied to all four columns of the dataframe. Note that ordering on the basis of our variable called `date` does not work in the way we want it to:

```
nums[order(date),]
```

```
        name       date    response treatment
53    rachel 01/08/2003 32.98792196         B
65    albert 02/06/2003 38.41979568         A
6        ann 02/07/2003  2.86983693         B
10    cecily 02/11/2003  6.81467570         A
4        ian 02/12/2003  2.09505949         A
29   michael 03/05/2003 15.59890900         B
67   william 03/09/2003 38.95014474         A
```

This is because of the format used for depicting the date is a character string in which the first characters are the day, then the month, then the year, so the dataframe has been sorted into alphabetical order, rather than date order as required. In order to sort by date we need first to convert our variable into date-time format using the `strptime` function (see p. 103 for details):

```
dates <- strptime(date,format="%d/%m/%Y")
dates
```

```
[1]   "2003-08-25"   "2003-05-21"   "2003-10-12"   "2003-12-02"   "2003-10-18"
[6]   "2003-07-02"   "2003-09-27"   "2003-06-05"   "2003-06-11"   "2003-11-02"
```

Note how `strptime` has produced a date object with year first, then a hyphen, then month, then a hyphen, then day, and this will sort into the desired sequence. We bind the new variable to the dataframe like this:

```
nums <- cbind(nums,dates)
```

Now that the new variable is in the correct format, the dates can be sorted correctly:

```
nums[order(dates),]
```

```
        name       date    response treatment      dates
49    albert 21/04/2003 30.66632632         A 2003-04-21
63     james 24/04/2003 37.04140266         A 2003-04-24
24      john 27/04/2003 12.70257306         A 2003-04-27
33   william 30/04/2003 18.05707279         B 2003-04-30
29   michael 03/05/2003 15.59890900         B 2003-05-03
71       ian 06/05/2003 39.97237868         A 2003-05-06
```

```
50    rachel 09/05/2003 30.81807436         B 2003-05-09
69 elizabeth 12/05/2003 39.39536726         B 2003-05-12
```

4.12 Using the `match` function in dataframes

The `worms` dataframe above contains fields of five different vegetation types:

```
unique(worms$Vegetation)
```

```
[1] Grassland Arable    Meadow    Scrub     Orchard
Levels: Arable Grassland Meadow Orchard Scrub
```

and we want to know the appropriate herbicides to use in each of the 20 fields. The herbicides are in a separate dataframe that contains the recommended herbicides for a much larger set of plant community types:

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

```
          Type   Herbicide
1    Woodland    Fusilade
2     Conifer    Weedwipe
3      Arable    Twinspan
4        Hill    Weedwipe
5     Bracken    Fusilade
6       Scrub    Weedwipe
7   Grassland    Allclear
8       Chalk    Vanquish
9      Meadow    Propinol
10       Lawn    Vanquish
11    Orchard    Fusilade
12      Verge    Allclear
```

The task is to create a vector of length 20 (one for every field in `worms`) containing the name of the appropriate herbicide. The first value needs to be Allclear because Nash's Field is grassland, and the second needs to be Twinspan because Silwood Bottom is arable, and so on. The first argument in match is `worms$Vegetation` and the second argument in match is `herbicides$Type`. The result of this match is used as a vector of subscripts to extract the relevant herbicides from `herbicides$Herbicide` like this:

```
herbicides$Herbicide[match(worms$Vegetation,herbicides$Type)]
```

```
[1]  Allclear Twinspan Allclear Propinol Weedwipe Allclear Allclear
[8]  Twinspan Fusilade Allclear Weedwipe Allclear Allclear Allclear
[15] Propinol Propinol Weedwipe Twinspan Allclear Weedwipe

Levels: Allclear Fusilade Propinol Twinspan Vanquish Weedwipe
```

You could add this information as a new column in the `worms` dataframe:

```
worms$hb <- herbicides$Herbicide[match(worms$Vegetation,herbicides$Type)]
```

or create a new dataframe called `recs` containing the herbicide recommendations:

```
recs <- data.frame(
      worms,hb=herbicides$Herbicide[match(worms$Vegetation,herbicides$Type)])
recs
```

```
          Field.Name Area Slope Vegetation Soil.pH  Damp Worm.density       hb
1        Nashs.Field  3.6    11  Grassland     4.1 FALSE            4 Allclear
2     Silwood.Bottom  5.1     2     Arable     5.2 FALSE            7 Twinspan
3      Nursery.Field  2.8     3  Grassland     4.3 FALSE            2 Allclear
4        Rush.Meadow  2.4     5     Meadow     4.9  TRUE            5 Propinol
5    Gunness.Thicket  3.8     0      Scrub     4.2 FALSE            6 Weedwipe
6           Oak.Mead  3.1     2  Grassland     3.9 FALSE            2 Allclear
7       Church.Field  3.5     3  Grassland     4.2 FALSE            3 Allclear
8            Ashurst  2.1     0     Arable     4.8 FALSE            4 Twinspan
9        The.Orchard  1.9     0    Orchard     5.7 FALSE            9 Fusilade
10     Rookery.Slope  1.5     4  Grassland     5.0  TRUE            7 Allclear
11       Garden.Wood  2.9    10      Scrub     5.2 FALSE            8 Weedwipe
12      North.Gravel  3.3     1  Grassland     4.1 FALSE            1 Allclear
13      South.Gravel  3.7     2  Grassland     4.0 FALSE            2 Allclear
14 Observatory.Ridge  1.8     6  Grassland     3.8 FALSE            0 Allclear
15        Pond.Field  4.1     0     Meadow     5.0  TRUE            6 Propinol
16      Water.Meadow  3.9     0     Meadow     4.9  TRUE            8 Propinol
17         Cheapside  2.2     8      Scrub     4.7  TRUE            4 Weedwipe
18        Pound.Hill  4.4     2     Arable     4.5 FALSE            5 Twinspan
19        Gravel.Pit  2.9     1  Grassland     3.5 FALSE            1 Allclear
20         Farm.Wood  0.8    10      Scrub     5.1  TRUE            3 Weedwipe
```

4.13 Merging two dataframes

Suppose we have two dataframes, the first containing information on plant life forms and the second containing information of time of flowering. We want to produce a single dataframe showing information on both life form and flowering time. Both dataframes contain variables for genus name and species name:

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

```
   Genus     species lifeform
1   Acer platanoides     tree
2   Acer    palmatum     tree
3  Ajuga     reptans     herb
4 Conyza sumatrensis   annual
5 Lamium       album     herb
```

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

```
      Genus       species flowering
1      Acer   platanoides       May
2     Ajuga       reptans      June
3  Brassica         napus     April
4 Chamerion angustifolium      July
5    Conyza     bilbaoana    August
6    Lamium         album   January
```

Because at least one of the variable names is identical in the two dataframes (in this case, two variables are identical, namely `Genus` and `species`) we can use the simplest of all merge commands:

```
merge(flowering,lifeforms)
```

```
   Genus     species flowering lifeform
1   Acer platanoides       May     tree
2  Ajuga     reptans      June     herb
3 Lamium       album   January     herb
```

The important point to note is that the merged dataframe contains only those rows which had *complete* entries in both dataframes. Two rows from the `lifeforms` database were excluded because there were no flowering time data for them (*Acer platanoides* and *Conyza sumatrensis*), and three rows from the `flowering` database were excluded because there were no life-form data for them (*Chamerion angustifolium*, *Conyza bilbaoana* and *Brassica napus*).

If you want to include all the species, with missing values (`NA`) inserted when flowering times or life forms are not known, then use the `all=T` option:

```
(both <- merge(flowering,lifeforms,all=T))
```

```
      Genus       species flowering lifeform
1      Acer   platanoides       May     tree
2      Acer      palmatum      <NA>     tree
3     Ajuga       reptans      June     herb
4  Brassica         napus     April     <NA>
5 Chamerion angustifolium      July     <NA>
6    Conyza     bilbaoana    August     <NA>
7    Conyza   sumatrensis      <NA>   annual
8    Lamium         album   January     herb
```

One complexity that often arises is that the same variable has *different names* in the two dataframes that need to be merged. The simplest solution is often to edit the variable names in your spreadsheet before reading them into R, but failing this, you need to specify the names in the first dataframe (known conventionally as the *x* dataframe) and the second dataframe (known conventionally as the *y* dataframe) using the `by.x` and `by.y` options in merge. We have a third dataframe containing information on the seed weights of all eight species, but the variable `Genus` is called `name1` and the variable `species` is called `name2`.

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

```
      name1         name2 seed
1      Acer   platanoides 32.0
2    Lamium         album 12.0
3     Ajuga       reptans  4.0
4 Chamerion angustifolium  1.5
5    Conyza     bilbaoana  0.5
6  Brassica         napus  7.0
7      Acer      palmatum 21.0
8    Conyza   sumatrensis  0.6
```

Just using `merge(both,seeds)` fails miserably: you should try it, to see what happens. We need to inform the `merge` function that `Genus` and `name1` are synonyms (different names for the same variable), as are `species` and `name2`.

```
merge(both,seeds,by.x=c("Genus","species"),by.y=c("name1","name2"))
```

```
      Genus       species flowering lifeform seed
1      Acer      palmatum      <NA>     tree 21.0
2      Acer   platanoides       May     tree 32.0
3     Ajuga       reptans      June     herb  4.0
4  Brassica         napus     April     <NA>  7.0
5 Chamerion angustifolium      July     <NA>  1.5
6    Conyza     bilbaoana    August     <NA>  0.5
7    Conyza   sumatrensis      <NA>   annual  0.6
8    Lamium         album   January     herb 12.0
```

Note that the variable names used in the merged dataframe are the names used in the *x* dataframe.

4.14 Adding margins to a dataframe

Suppose we have a dataframe showing sales by season and by person:

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

```
             name spring summer autumn winter
1      Jane.Smith     14     18     11     12
2    Robert.Jones     17     18     10     13
3     Dick.Rogers     12     16      9     14
4 William.Edwards     15     14     11     10
5     Janet.Jones     11     17     11     16
```

We want to add margins to this dataframe showing departures of the seasonal means from the overall mean (as an extra row at the bottom) and departures of the people's means (as an extra column on the right). Finally, we want the sales in the body of the dataframe to be represented by departures from the overall mean.

```
people <- rowMeans(frame[,2:5])
people <- people-mean(people)
people
```

```
[1]  0.30  1.05 -0.70 -0.95  0.30
```

It is very straightforward to add a new column to the dataframe using `cbind`:

```
(new.frame <- cbind(frame,people))
```

```
             name spring summer autumn winter people
1      Jane.Smith     14     18     11     12   0.30
2    Robert.Jones     17     18     10     13   1.05
3     Dick.Rogers     12     16      9     14  -0.70
4 William.Edwards     15     14     11     10  -0.95
5     Janet.Jones     11     17     11     16   0.30
```

Robert Jones is the most effective sales person (+ 1.05) and William Edwards is the least effective (–0.95). The column means are calculated in a similar way:

```
seasons <- colMeans(frame[,2:5])
seasons <- seasons-mean(seasons)
seasons
```

```
spring summer autumn winter
0.35   3.15  -3.05  -0.45
```

Sales are highest in summer (+ 3.15) and lowest in autumn (–3.05).

Now there is a hitch, however, because there are only four column means but there are six columns in `new.frame`, so we cannot use `rbind` directly. The simplest way to deal with this is to make a copy of one of the rows of the new dataframe

```
new.row <- new.frame[1,]
```

and then edit this to include the values we want: a label in the first column to say 'seasonal means' then the four column means, and then a zero for the grand mean of the effects:

```
new.row[1] <- "seasonal effects"
new.row[2:5] <- seasons
new.row[6] <- 0
new.row
```

```
              name spring summer autumn winter people
1 seasonal effects   0.35   3.15  -3.05  -0.45      0
```

Now we can use `rbind` to add our new row to the bottom of the extended dataframe:

```
(new.frame <- rbind(new.frame,new.row))
```

```
              name spring summer autumn winter people
1       Jane.Smith  14.00  18.00  11.00  12.00   0.30
2     Robert.Jones  17.00  18.00  10.00  13.00   1.05
3      Dick.Rogers  12.00  16.00   9.00  14.00  -0.70
4  William.Edwards  15.00  14.00  11.00  10.00  -0.95
5      Janet.Jones  11.00  17.00  11.00  16.00   0.30
6 seasonal effects   0.35   3.15  -3.05  -0.45   0.00
```

The last task is to replace the counts of sales in the dataframe `new.frame[1:5,2:5]` by departures from the overall mean sale per person per season (the grand mean, `gm` = 13.45). We need to use `unlist` to stop R from estimating a separate mean for each column, then create a vector of length 4 containing repeated values of the grand mean (one for each column of sales). Finally, we use `sweep` to subtract the grand mean from each value:

```
gm <- mean(unlist(new.frame[1:5,2:5]))
gm <- rep(gm,4)
new.frame[1:5,2:5] <- sweep(new.frame[1:5,2:5],2,gm)
new.frame
```

```
              name spring summer autumn winter people
1       Jane.Smith   0.55   4.55  -2.45  -1.45   0.30
2     Robert.Jones   3.55   4.55  -3.45  -0.45   1.05
3      Dick.Rogers  -1.45   2.55  -4.45   0.55  -0.70
4  William.Edwards   1.55   0.55  -2.45  -3.45  -0.95
```

```
5      Janet.Jones  -2.45   3.55  -2.45   2.55   0.30
6 seasonal effects   0.35   3.15  -3.05  -0.45   0.00
```

To complete the table we want to put the grand mean in the bottom right-hand corner:

```
new.frame[6,6] <- gm[1]
new.frame
```

```
              name spring summer autumn winter people
1       Jane.Smith   0.55   4.55  -2.45  -1.45   0.30
2     Robert.Jones   3.55   4.55  -3.45  -0.45   1.05
3      Dick.Rogers  -1.45   2.55  -4.45   0.55  -0.70
4  William.Edwards   1.55   0.55  -2.45  -3.45  -0.95
5      Janet.Jones  -2.45   3.55  -2.45   2.55   0.30
6 seasonal effects   0.35   3.15  -3.05  -0.45  13.45
```

The best per-season performance was shared by Jane Smith and Robert Jones who each sold 4.55 units more than the overall average in summer.

4.15 Summarizing the contents of dataframes

The usual function to obtain cross-classified summary functions like the mean or median for a single vector is `tapply` (p. 245), but there are three useful functions for summarizing whole dataframes:

- `summary` summarize all the contents of all the variables;
- `aggregate` create a table after the fashion of `tapply`;
- `by` perform functions for each level of specified factors.

Use of `summary` and `by` with the worms database was described on p. 163. The `aggregate` function is used like `tapply` to apply a function (`mean` in this case) to the levels of a specified categorical variable (`Vegetation` in this case) for a specified range of variables (`Area`, `Slope`, `Soil.pH` and `Worm.density`) which are specified using their subscripts as a column index, `worms[,c(2,3,5,7)]`:

```
aggregate(worms[,c(2,3,5,7)],by=list(veg=Vegetation),mean)
```

```
        veg     Area    Slope  Soil.pH Worm.density
1    Arable 3.866667 1.333333 4.833333     5.333333
2 Grassland 2.911111 3.666667 4.100000     2.444444
3    Meadow 3.466667 1.666667 4.933333     6.333333
4   Orchard 1.900000 0.000000 5.700000     9.000000
5     Scrub 2.425000 7.000000 4.800000     5.250000
```

The `by` argument needs to be a `list` even if, as here, we have only one classifying factor. Here are the aggregated summaries cross-classified by `Vegetation` and `Damp`:

```
aggregate(worms[,c(2,3,5,7)],by=list(veg=Vegetation,d=Damp),mean)
```

```
        veg     d     Area    Slope  Soil.pH Worm.density
1    Arable FALSE 3.866667 1.333333 4.833333     5.333333
2 Grassland FALSE 3.087500 3.625000 3.987500     1.875000
```

```
3   Orchard FALSE 1.900000 0.000000 5.700000     9.000000
4     Scrub FALSE 3.350000 5.000000 4.700000     7.000000
5 Grassland  TRUE 1.500000 4.000000 5.000000     7.000000
6    Meadow  TRUE 3.466667 1.666667 4.933333     6.333333
7     Scrub  TRUE 1.500000 9.000000 4.900000     3.500000
```

Note that this summary is unbalanced because there were no damp arable or orchard sites and no dry meadows.