4 Data: Descriptive Statistics and Tabulation

WHAT YOU WILL LEARN IN THIS CHAPTER:

- How to summarize data samples
- How to use cumulative statistics
- How to create summary tables
- How to cross-tabulate
- How to test for different object types

Important elements in data analysis include summary and descriptive statistics. These provide a shorthand way of describing and summarizing your data, which is important in pointing you towards the correct analytical procedure and helping you understand your data. There are three main ways you can describe or summarize your data:

- Summary statistics
- Tabulation
- Graphical

In this chapter you will learn about using summary statistics to provide a shorthand way of describing your data as opposed to merely listing the contents. You will also look at tabulation as a method to create summaries. Tables can split your data into manageable chunks that show you patterns that you would otherwise miss. Producing a graphical summary of your data is also important because a visual impression can convey more to a reader than numerical values; these are the subjects of Chapter 5.

SUMMARY COMMANDS

An essential starting point with any set of data is to get an overview of what you are dealing with. There are a few ways to go about doing this. You might start by using the `ls()` command to see what named objects you have. You can then type the name of one of the objects to view its contents. However, if the object contains a lot of data, the display may be quite long (and somewhat overwhelming); you will want a more concise method to examine objects. You could use the `str()` command, which shows you something about the structure of the data. Take, for instance, the following data frame called `grass`. This contains two columns: one is titled `rich` and relates to the number of plant species found in quadrats and the other is titled `graze` and relates to the mowing treatment of the site:

```
> grass
  rich graze
1   12   mow
2   15   mow
3   17   mow
4   11   mow
5   15   mow
6    8 unmow
7    9 unmow
8    7 unmow
9    9 unmow
```

In this case, there are not too many observations so you can easily see all the data. If you use the `str()` command like so, you see a more concise summary of the `grass` object:

```
> str(grass)
'data.frame': 9 obs. of  2 variables:
 $ rich : int  12 15 17 11 15 8 9 7 9
 $ graze: Factor w/ 2 levels "mow","unmow": 1 1 1 1 1 2 2 2 2
```

However, the `str()` command is designed to help you examine the structure of a data object rather than providing a statistical summary. By contrast, the `summary()` command is designed to give a quick statistical summary of data objects. The output you get depends on the object you are looking at. In this case there is a data frame so each column is summarized:

```
> summary(grass)
      rich          graze
 Min.   : 7.00   mow  :5
 1st Qu.: 9.00   unmow:4
 Median :11.00
 Mean   :11.44
 3rd Qu.:15.00
 Max.   :17.00
```

Here you see some basic statistics for the numeric column; you can see the largest and smallest values as well as central (median and mean) measures. The second column does not contain numbers, so you see a list of the different factors along with a count for each. Here you see five observations for the `mow` treatment but only four replicates for the `unmow` treatment.

If your data contain character items that are in quotes, they are treated as standard characters rather than as factors. When you attempt a `summary()` you get slightly different results. In the following example, you can see a simple vector in two forms:

```
> graze
[1] "mow"   "mow"   "mow"   "mow"   "mow"   "unmow" "unmow" "unmow" "unmow"
> summary(graze)
   Length     Class      Mode 
        9 character character 
```

Here, the data are true characters (notice the quotes) and the summary merely tells you that there are nine of them. Earlier, you created a data frame using these data alongside a vector of numbers (the `rich` data). The resulting data frame converted the character items into factors. They are still characters (as opposed to numbers), but are handled differently. In most statistical analysis you want your character data as factors. If you extract the data from the data frame and run a `summary()`, you get a different result:

```
> grass$graze
[1] mow   mow   mow   mow   mow   unmow unmow unmow unmow
Levels: mow unmow
> summary(grass$graze)
  mow unmow 
    5     4 
```

The `summary()` command is therefore more useful; you can see two factors and the number of observations (replicates) for each.

The `summary()` command works for both matrix and data frame objects by summarizing the columns rather than the rows. You can think of vectors as individual columns, and the `summary()` command works on vectors in this fashion. When it comes to list objects though, things are not so simple. In the following example there is a list that comprises two numeric vectors of unequal length:

```
> grass.l
$mow
[1] 12 15 17 11 15

$unmow
[1] 8 9 7 9

> summary(grass.l)
      Length Class  Mode   
mow   5      -none- numeric
unmow 4      -none- numeric
```

The `summary()` command looks at the object as a series of columns, but because a list does not have any columns, you get a simpler result. You can get the "proper" summary by applying the command to each item in the list. However, you do need to specify the name exactly:

```
> grass.l$mow
[1] 12 15 17 11 15
> summary(grass.l$mow)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max. 
     11      12      15      14      15      17 
```

In the preceding example you see that by adding the `$` you can extract an item from the list and the `summary()` command works as you expect.

It is sometimes useful to see what the columns are called, especially when the data are in a large data frame. Similarly, on some occasions you need a reminder of the row names. You can use the `names()` command and its variants, which you met previously. Table 4-1 is a reminder of these options.

TABLE 4-1: Summary of Commands to Find or Add Names to Rows and Columns of Data Objects

COMMAND	EXPLANATION
`names()`	Works on list or data frame objects. Gets or sets names for columns of a data frame or the elements of a list.
`row.names()`	Works on matrix or data frame objects.
`rownames()`	Works on matrix or data frame objects.
`colnames()`	Works on matrix or data frame objects.
`dimnames()`	Gets row and column names for matrix or data frame objects.

SUMMARIZING SAMPLES

In Chapter 2, you looked at some simple math. In general you were operating on simple numbers; that is, you looked at single values as opposed to a set of numbers that formed a sample. When you have repeated measurements (that is, a sample) you usually want to summarize the data by showing things like the average. In R you have a variety of commands that operate on samples; these samples of data might be individual vectors, or they may be columns in a data frame or part of a matrix or list.

Summary Statistics for Vectors

The simplest data object you will encounter is the vector. A vector is a single column of values—a one-dimensional object. There are a variety of simple summary statistics that can be applied to a vector of numbers, some of which you will meet shortly. In general there are two kinds of summary commands:

- Commands that produce a single value as a result
- Commands that produce multiple values as a result

The following sections deal with each of these kinds of summary commands.

Summary Commands With Single Value Results

You can use several commands to help you summarize simple numeric data. Table 4-2 shows some of the commands that produce a single value as their result.

TABLE 4-2: Commands that Produce a Single Value as a Summary Statistic

COMMAND	EXPLANATION
`max(x, na.rm = FALSE)`	Shows the maximum value. By default `NA` values are not removed. A value of `NA` is considered the largest unless `na.rm = TRUE` is used.
`min(x, na.rm = FALSE)`	Shows the minimum value in a vector. If there are `NA` values, this returns a value of `NA` unless `na.rm = TRUE` is used.
`length(x)`	Gives the length of the vector and includes any `NA` values. The `na.rm =` instruction does not work with this command.
`sum(x, na.rm = FALSE)`	Shows the sum of the vector elements.
`mean(x, na.rm = FALSE)`	Shows the arithmetic mean.
`median( x, na.rm = FALSE)`	Shows the median value of the vector.
`sd(x, na.rm = FALSE)`	Shows the standard deviation.
`var(x, na.rm = FALSE)`	Shows the variance.
`mad(x, na.rm = FALSE)`	Shows the median absolute deviation.

In the following activity you will examine some of the simple summary statistics on some numerical vectors.

TRY IT OUT Using Summarizing Commands on a Sample

Available for download on Wrox.com

Use the `data2` and `unmow` data objects from the `Beginning.RData` file for this activity. You will be using some simple summarizing commands on these data.

1. Type the name of the object you will be examining, in this case `data2`:

```
> data2
 [1] 3 5 7 5 3 2 6 8 5 6 9 4 5 7 3 4
```

2. Display the average of the sample as a mean value:

```
> mean(data2)
[1] 5.125
```

3. Now determine the largest value in the sample:

```
> max(data2)
[1] 9
```

4. Next determine the smallest value in the sample:

```
> min(data2)
[1] 2
```

5. Look now at how many items are in the sample:

```
> length(data2)
[1] 16
```

6. Now look at a different data sample, the `unmow` object:

```
> unmow
[1]  8  9  7  9 NA
```

7. Work out the standard deviation of the complete `unmow` sample:

```
> sd(unmow)
[1] NA
```

8. Calculate the standard deviation but remove `NA` items with an additional instruction:

```
> sd(unmow, na.rm = TRUE)
[1] 0.9574271
```

How It Works

The various commands operate on the vector of values to return a simple result. However, if `NA` items are present the final value will also be `NA`. For most commands you can ensure that any `NA` items are ignored by adding the `na.rm = TRUE` instruction to the command. Now you get a “proper” result.

NOTE *Many summarizing commands use the* `na.rm` *instruction to eliminate* `NA` *items from the summary. However, this is not universal, the* `length()` *command does not use* `na.rm` *for example.*

Omitting NA Items

The `length()` command does not use the `na.rm` instruction so you need a way to overcome this; fortunately there is a solution. You can use the `na.omit()` command to strip out `NA` items. Essentially, you use this to temporarily remove `NA` items like so:

```
> length(na.omit(unmow))
[1] 4
```

Altering Sample Length

You can use the `length()` command to alter the length of a vector by setting it to a numeric value. The vector is shortened if your value is less than the current setting. If the setting is longer than the current setting, `NA` items are added to make it the required length. This can be useful if you want to make a data frame from vectors of unequal length; you can set the length of all vectors to be the same as the longest one:

```
> unmow
[1]  8  9  7  9 NA
> length(unmow)
[1] 5
```

```
> length(unmow) = 4
> unmow
[1] 8 9 7 9
> length(unmow) = 6
> unmow
[1]  8  9  7  9 NA NA
```

In the preceding example, the original vector contained five elements but one was `NA`, producing a length of five. When you set the length to four, the last element (`NA` in the preceding example) is stripped off. When you reset the length to six, you get an additional two `NA` items at the end.

Summary Commands With Multiple Results

So far the commands you have used have produced a single value as a result. However, many commands produce several values. When you looked at some of the mathematical operations in Chapter 2, you applied your math to simple numbers. If you apply a math function to a vector, you get an answer back for each element of the vector like so:

```
> data2
 [1] 3 5 7 5 3 2 6 8 5 6 9 4 5 7 3 4
> log(data2)
 [1] 1.0986123 1.6094379 1.9459101 1.6094379 1.0986123 0.6931472 1.7917595
 [8] 2.0794415 1.6094379 1.7917595 2.1972246 1.3862944 1.6094379 1.9459101
[15] 1.0986123 1.3862944
```

Of course, the `log()` command is not one you would normally think of as a summary command. Summary commands that do produce multiple results are illustrated here:

```
> summary(data2)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  2.000   3.750   5.000   5.125   6.250   9.000

> quantile(data2)
  0%  25%  50%  75% 100%
2.00 3.75 5.00 6.25 9.00

> fivenum(data2)
[1] 2.0 3.5 5.0 6.5 9.0
```

You have already met the basic `summary()` command. For this simple numeric vector you get two measures of centrality: the mean and median. You also get the extremes as well as the inter-quartile values. The `quantile()` command shows the quartiles by default; that is, the 0%, 25%, 50%, 75% and 100% quantiles. However, you can select other quantiles. The command allows other instructions as follows:

```
quantile(x, probs = seq(0, 1, 0.25), na.rm = FALSE, names = TRUE)
```

The `x` part is the data object you wish to examine. The `probs =` instruction enables you to select one or several quantiles to display, defaulting to 0, 0.25, and so on as you saw in the preceding example. This is what the `seq(0, 1, 0.25)` command is doing; setting a start of 0, an end of 1, and a step of 0.25. This is the same as `c(0, 0.25, 0.5, 0.75, 1)`. The `names =` instruction tells R if it should display the name of the quantiles produced. In the following activity you examine some of the options for the `quantile()` command for yourself.

TRY IT OUT Using the quantile() Command

Available for download on Wrox.com

Use the `data2` and `unmow` data objects from the `Beginning.RData` file for this activity. You will use the `quantile()` command on these data.

1. Start by looking at the sample vector, `data2`.

```
> data2
 [1] 3 5 7 5 3 2 6 8 5 6 9 4 5 7 3 4
```

2. Now look at the 20% quantile:

```
> quantile(data2, 0.2)
20%
  3
```

3. Next pick out three quantiles, 20%, 50%, and 80%:

```
> quantile(data2, c(0.2, 0.5, 0.8))
20% 50% 80%
  3   5   7
```

4. Now try some quantiles in non-numeric order:

```
> quantile(data2, c(0.5, 0.75, 0.25))
 50%  75%  25%
5.00 6.25 3.75
```

5. Select some quantiles but suppress the headings:

```
> quantile(data2, c(0.2, 0.5, 0.8), names = F)
[1] 3 5 7
```

6. Look at a new data object that contains `NA` items:

```
> unmow
[1]  8  9  7  9 NA NA
```

7. Display the basic quantiles for the new sample:

```
> quantile(unmow)
Error in quantile.default(unmow) :
  missing values and NaN's not allowed if 'na.rm' is FALSE
```

8. Remove the effect of the `NA` items using the `na.rm` instruction:

```
> quantile(unmow, na.rm = T)
  0%  25%  50%  75% 100%
7.00 7.75 8.50 9.00 9.00
```

How It Works

The `quantile()` command produces multiple results but you can alter the default to produce quantiles for a single probability or several (in any order). The names of the quantiles selected are displayed as percentage labels but you can suppress this using the `names = FALSE` instruction. If the data contain `NA` items, you must remove them using the `na.rm = TRUE` instruction, otherwise you get an error message.

The fivenum() command produces a similar result to quantile(), but in this case 25% and 75% quantiles (the inter-quartiles) are replaced by the lower and upper hinge values. These are similar to the quantiles and for samples with odd-number lengths they are the same:

```
> dat
[1] 1 2 3 4 5 6
> quantile(dat)
  0%  25%  50%  75% 100%
1.00 2.25 3.50 4.75 6.00

> fivenum(dat)
[1] 1.0 2.0 3.5 5.0 6.0
```

By default NA items are removed, but you could include the NA using na.rm = FALSE as an instruction in the command if you want to keep them in (in which case all the quantiles would be NA).

Cumulative Statistics

Cumulative statistics are those that are applied sequentially to a series of values. For example, you may want to track the interest received on an investment. If your data are the interest payments received then the cumulative sum would give you a running total. You can think of the commands that calculate cumulative statistics as being in one of two forms:

- Simple cumulative commands
- Complex cumulative commands

You will look at both types in this section. Simple commands require only the name of the data object. For complex commands you have to create more complicated instructions to produce the desired result.

Simple Cumulative Commands

Table 4-3 shows several simple commands that you can use that return cumulative values.

TABLE 4-3: Commands that Produce Cumulative Values

COMMAND	EXPLANATION
cumsum(x)	The cumulative sum of a vector
cummax(x)	The cumulative maximum value
cummin(x)	The cumulative minimum value
cumprod(x)	The cumulative product

In the following activity you try out some cumulative statistics for yourself.

TRY IT OUT Cumulative Statistics

Available for download on Wrox.com

Use the data2 and data5 data objects from the Beginning.RData file for this activity; this contains the data objects that you will need to produce cumulative statistics.

1. Start by looking at a simple numerical vector, data2.

```
> data2
 [1] 3 5 7 5 3 2 6 8 5 6 9 4 5 7 3 4
```

2. Determine the cumulative sum of these data:

```
> cumsum(data2)
 [1]  3  8 15 20 23 25 31 39 44 50 59 63 68 75 78 82
```

3. Now look at the cumulative maximum value of the sample:

```
> cummax(data2)
 [1] 3 5 7 7 7 7 7 8 8 8 9 9 9 9 9 9
```

4. Try looking at the cumulative minimum:

```
> cummin(data2)
 [1] 3 3 3 3 3 2 2 2 2 2 2 2 2 2 2 2
```

5. Now look at the cumulative product of the sample:

```
> cumprod(data2)
 [1]           3          15         105         525        1575        3150
 [7]       18900      151200      756000     4536000    40824000   163296000
[13]   816480000  5715360000 17146080000 68584320000
```

6. Try a cumulative command on a vector of character data (for example, data5):

```
> data5
 [1] "Jan" "Feb" "Mar" "Apr" "May" "Jun" "Jul" "Aug" "Sep" "Oct" "Nov" "Dec"
> cummax(data5)
 [1] NA NA NA NA NA NA NA NA NA NA NA NA
Warning message:
NAs introduced by coercion
```

7. Now look at a data sample that includes NA items:

```
> dat.na
[1]  2  5  4 NA  7  3  9 NA 12
```

8. Try a cumulative command on these data:

```
> cumprod(dat.na)
[1]  2 10 40 NA NA NA NA NA NA
```

How It Works

The cumulative commands produce the expected result until you try them on a vector of character data. If you try these on character data you get an error, and your “result” is a list of NA items. If your numeric vector contains any NA items, the commands will “work” up to the first NA item, and subsequently you get NA.

Complex Cumulative Commands

You can use cumulative commands in combination with others to produce additional useful measures; for example, the running mean. The basic arithmetic mean is the sum divided by the number of observations. You can get the cumulative sum, but you also need the cumulative number of observations; you can use the `seq()` command to help. If you have a sample of numeric values you can create an index like so:

```
> data2
 [1] 3 5 7 5 3 2 6 8 5 6 9 4 5 7 3 4
> seq(along = data2)
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16
```

In this case you have created a simple index for your sample; you have 16 items. You can also use a "quick" version like so:

```
> seq_along(data2)
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16
```

USING THE SEQ() COMMAND

The `seq()` command can be used in more than one way. The main purpose of the command is to generate sequences of values. You can specify the start point, end point, and interval for the sequence you require like so:

```
> seq(from = 1, to = 10, by = 2)
[1] 1 3 5 7 9
```

The command can be abbreviated, and the following produces the same result:

```
> seq(1, 10, 2)
[1] 1 3 5 7 9
```

If you use the `along =` instruction you can create an index for a vector:

```
> seq(along = data2)
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16
```

However, if you omit the `along =` part and specify the vector name you get the same result:

```
> seq(data2)
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16
```

The `seq_along()` command is thus the equivalent of using the `along =` instruction in the regular `seq()` command.

If you now combine the cumulative sum and "how far along" you are, you can make a running mean like so:

```
> cumsum(data2) / seq(along = data2)
 [1] 3.000000 4.000000 5.000000 5.000000 4.600000 4.166667 4.428571 4.875000
 [9] 4.888889 5.000000 5.363636 5.250000 5.230769 5.357143 5.200000 5.125000
```

For other cumulative statistics you may have to be a bit more creative. For example, you might want to use a running median. None of the cumulative commands can really help you here. The answer is to use the `seq_along()` command as an index and container and determine your median as you step along. The following example and subsequent steps illustrate the process:

```
> data2
 [1] 3 5 7 5 3 2 6 8 5 6 9 4 5 7 3 4
> md = seq_along(data2)
> md
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16
> for(i in 1:length(md)) md[i] = median(data2[1:i])
> md
 [1] 3 4 5 5 5 4 5 5 5 5 5 5 5 5 5 5
```

1. Begin by creating an item called `md`, which acts as the repository for your final result. In the meantime you can also use this as an index to help you generate the median.
2. Use the `for()` command to make your `median()` work 16 times (in this case). The first part of the command tells R how many times to set the temporary item `i`.
3. Now use the `median()` command to place a value into your result object (called `md`); do this 16 times and you end up with a final vector of 16 values, which represents the running median.
4. The `for()` command has two main parts: the first part (in the parentheses) is where you set the number of times you want to repeat your `expression` (the second part). It is common to use `i` as a variable name in `for()` commands (think of it as `i` for index), but there is no reason not to use anything you like!
5. Separate the variable from the sequence you want to use with the word `in`. Finally, create an `expression` that uses your repeating variable. The following shows the general form of the `for()` command:

   ```
   for(var in seq) expression
   ```

You could easily replace the `median` part in the example to another command and produce a running *something*. In the following example the `sd()` command is used rather than `median()` to determine the running standard deviation:

```
> md = seq_along(data2)
> for(i in 1:length(md)) md[i] = sd(data2[1:i])
> md
 [1]       NA 1.414214 2.000000 1.632993 1.673320 1.834848 1.812654 2.100170
 [9] 1.964971 1.885618 2.157440 2.094365 2.006400 1.984833 2.007130 1.962142
```

In this example the first item in the result is `NA` because you cannot calculate a standard deviation of a single value. You look at the `for()` command again in Chapter 10, where you learn a bit more about customizing functions and creating simple scripts and programming.

Summary Statistics for Data Frames

So far you have looked to summarize a single vector of data, but there may be times when you want to summarize a more complicated object. Some of the commands you have used already will work on more complex objects, and some need a bit of persuading.

Generic Summary Commands for Data Frames

Table 4-4 gives a quick guide to the results expected for some of the generic summary commands you met previously; it is not complete but it covers the more useful summary commands.

TABLE 4-4: Summary Commands that Can be Applied to Data Frames

COMMAND	EXPLANATION
`max(frame)`	The largest value in the entire data frame
`min(frame)`	The smallest value in the entire data frame
`sum(frame)`	The sum of the entire data frame
`fivenum(frame)`	The Tukey summary values for the entire data frame
`length(frame)`	The number of columns in the data frame
`summary(frame)`	Gives summary for each column

The list of summary commands that will work on a data frame is quite short. You can always extract a single vector from your data frame and perform a summary of some sort on that. This approach will not work for the rows of a data frame though. In general it is better to use more specialized commands when dealing with the rows and columns of data frames. You will meet these commands in the following sections.

Special Row and Column Summary Commands

Two summary commands are designed especially for row data—`rowMeans()` and `rowSums()`:

```
> rowMeans(fw)
     Taw Torridge     Ouse      Exe      Lyn    Brook    Ditch      Fal
     5.5     14.0     10.0      5.5     14.0     24.5     26.5     40.5
> rowSums(fw)
     Taw Torridge     Ouse      Exe      Lyn    Brook    Ditch      Fal
      11       28       20       11       28       49       53       81
```

In the example here each row has a row name so these are displayed. If you did not have the row names the values for the various rows would appear as a simple vector of values like so:

```
> rowSums(mf)
 [1] 274.25 262.15 215.75 240.95 227.95 228.75 197.85 264.75 247.95 262.35 267.35
[12] 264.35 259.05 245.85 229.75 247.45 275.35 253.05 201.25 295.05 275.55 176.85
[23] 204.95 218.85 208.75
```

Corresponding `colSums()` and `colMeans()` commands function in the same manner. In the following example you see the `mean()` and `colMeans()` commands compared:

```
> colMeans(mf)
    len      sp     alg     no3     bod
 19.640  15.800  58.400   2.046 145.960
> mean(mf)
    len      sp     alg     no3     bod
 19.640  15.800  58.400   2.046 145.960
```

You can see that essentially you get the same display/result. These commands also use the `na.rm` instruction, and by default this is set to `FALSE`. If you want to ensure that `NA` items are removed, you add `na.rm = TRUE` as an instruction in the command.

The apply() Command for Summaries on Rows or Columns

The `colMeans()` and `rowSums()` commands are designed as quick alternatives to a more general command, `apply()`. The `apply()` command enables you to apply a function to rows or columns of a matrix or data frame. The general form of the command is like so:

```
apply(X, MARGIN, FUN, ...)
```

In this command the `MARGIN` is either 1 or 2, where 1 is for rows and 2 is for columns. You replace the `FUN` part with your command (the function you want to apply) and you can also add additional instructions if they are appropriate to the command/function you are applying. For example, you might add the `na.rm = TRUE` instruction:

```
> apply(fw, 1, mean, na.rm = TRUE)
     Taw Torridge     Ouse      Exe      Lyn    Brook    Ditch      Fal
     5.5     14.0     10.0      5.5     14.0     24.5     26.5     40.5
```

In this case you see that the row names of the original data frame are displayed. If your data frame had no set row names, you would simply see your result as a vector of values like so:

```
> apply(mf, 1, median, na.rm = TRUE)
 [1] 20 21 22 23 21 21 19 16 16 21 21 26 21 20 19 18 17 19 21 21 22 25 24 23 22
```

Summary Statistics for Matrix Objects

A matrix looks like a frame but is not; in effect, the data are a single vector that happens to be split into rows and columns. In the following example you see a matrix comprised of some numeric values relating to observations of some common British birds in various habitats:

```
> bird
              Garden Hedgerow Parkland Pasture Woodland
Blackbird         47       10       40       2        2
Chaffinch         19        3        5       0        2
Great Tit         50        0       10       7        0
House Sparrow     46       16        8       4        0
Robin              9        3        0       0        2
Song Thrush        4        0        6       0        0
```

You cannot extract parts of a matrix using `$` like you could with a data frame, but you can use the square brackets to retrieve information about any row or column:

```
> mean(bird[,2])
[1] 5.333333
> mean(bird[2,])
[1] 5.8
```

The first example returns the mean for the second column, whilst the next example returns the mean for the second row. You can also use the `colMeans()` and `rowSums()` commands like you used before:

```
> colSums(bird)
  Garden Hedgerow Parkland  Pasture Woodland
     175       32       69       13        6
```

```
> rowMeans(bird)
    Blackbird     Chaffinch     Great Tit House Sparrow         Robin
         20.2           5.8          13.4          14.8           2.8
  Song Thrush
          2.0
```

The `apply()` command also works equally well for a matrix as it does for data frame objects, like so:

```
>apply(bird, 2, median)
  Garden Hedgerow Parkland  Pasture Woodland
    32.5      3.0      7.0      1.0      1.0
```

In this case you extract the median values for the columns of the matrix. You can also choose certain elements of the result to display by appending square brackets after the command, as shown in the following example:

```
> apply(bird,1,median)[1:2]
Blackbird Chaffinch
       10         3

> apply(bird,1,median)[c(1,2,4)]
    Blackbird     Chaffinch House Sparrow
           10             3             8

> apply(bird,1,median)[c(1,2,'Robin')]
  NA>   NA> Robin
   NA    NA     2

> apply(bird,1,median)[c('Blackbird','Robin')]
Blackbird     Robin
       10         2
```

In the first example you display only the first and second items. In the next example you select the first, second, and fourth items. The third example shows that you cannot mix numbers and text. In the final example you select the column results you want using their column names (in quotes).

Summary Statistics for Lists

List objects do not work in quite the same manner as matrix or data frame objects. As with most of the summary commands that you have met, many will simply fail to work, so a different approach is needed. In the following example you have a simple list that is comprised of two vectors of numbers that are unequal in length:

```
> grass.l
$mow
[1] 12 15 17 11 15

$unmow
[1] 8 9 7 9

> summary(grass.l)
      Length Class  Mode
mow   5      -none- numeric
unmow 4      -none- numeric
```

```
> mean(grass.l)
[1] NA
Warning message:
In mean.default(grass.l) : argument is not numeric or logical: returning NA
> sum(grass.l)
Error in sum(grass.l) : invalid 'type' (list) of argument
> length(grass.l)
[1] 2
```

The only useful result you get here is the `length()` command, which confirms that you have two elements in your list. You could examine each element of the list in turn by using the `$` syntax like so:

```
> mean(grass.l$mow)
[1] 14
> max(grass.l$unmow)
[1] 9
```

Using `$` is fine, but it's quite tedious if you have more than one or two elements to consider. It is also not generalized enough; it would be better to have a method that did not rely on individual items being named (other than the list object). Instead, you can use a special version of the `apply()` command that works specifically on list objects. The command is `lapply()`; think of it as short for "list apply." This is easy enough to use—you simply name the list and the function you want to apply to each list element like so:

```
> lapply(grass.l, mean, na.rm = TRUE)
$mow
[1] 14
$unmow
[1] 8.25
```

You are still able to add extra instructions to the command; in this example you ensure that `NA` items are removed before the `mean()` command is applied. The result you get back is in the form of a list pretty much like the original object. You can change this to produce a "nicer" output using a variant of the command `sapply()` like so.

```
> sapply(grass.l, mean, na.rm = TRUE)
  mow unmow
14.00  8.25
```

The resulting output is in fact a matrix. This enables you to undertake other manipulations because a matrix object is a bit easier to deal with than a list. If you want to carry out further manipulations on the result, make an object to hold the result:

```
> grass.mn = sapply(grass.l, mean, na.rm = TRUE)
```

Now you have a new object that can be used later.

SUMMARY TABLES

Just as you did with the vector, matrix, list, and data frame objects in Chapter 3, you can manipulate, alter, and produce table objects using the `table()` command to summarize a data sample. Using this command you can create a few special kinds of table objects, including contingency tables and complex (flat) contingency tables. A contingency table is particularly useful when you have a large number of observations and you want to condense the data into a smaller format. A complex (flat) table is a type of contingency table that is useful when creating just one single table as opposed to multiple ones. Additionally, you can use cross-tabulation to reassemble data into a tabular format as necessary. This section covers how to work with table objects in all of these capacities.

Making Contingency Tables

A contingency table is a way to redraw data and assemble it into a table that shows the layout of the original data in a manner that allows the reader to gain an overall summary of the original data. You can create contingency tables using the `table()` command. The command can handle data in simple vectors or more complex matrix and data frame objects, as you see shortly. The more complex the original data, the more complex the resulting contingency table will be.

Creating Contingency Tables from Vectors

The simplest data object from which you can create a contingency table is a vector. In the following example you have a simple numeric vector of values:

```
> data2
 [1] 3 5 7 5 3 2 6 8 5 6 9 4 5 7 3 4
> table(data2)
data2
2 3 4 5 6 7 8 9
1 3 2 4 2 2 1 1
```

Here you use the `table()` command to organize the data into a simple contingency table. This table shows you how many items in the data match up to the various integer values; you can see that there are three 3s, for example, but only a single 8. You can visualize this better, perhaps, if you rewrite the data in numerical order:

```
> sort(data2)
 [1] 2 3 3 3 4 4 5 5 5 5 6 6 7 7 8 9
```

Here the `sort()` command is used to reorder the data values; if you compare this to the table you just created you can see more clearly what the `table()` command has done. You can use the `table()` command on character data too; in the following example you have a simple vector of labels:

```
> graze
[1] "mow"   "mow"   "mow"   "mow"   "mow"   "unmow" "unmow" "unmow" "unmow"
> table(graze)
graze
  mow unmow
    5     4
```

You can see from the table that there are five items in the `mow` treatment and four in the `unmow` treatment.

Creating Contingency Tables from Complicated Data

The numeric data that go with the labels from the preceding section's example are assembled into a data frame like so:

```
> grass
```

```
  rich graze
1   12   mow
2   15   mow
3   17   mow
4   11   mow
5   15   mow
6    8 unmow
7    9 unmow
8    7 unmow
9    9 unmow
```

If you use the `table()` command on these data you get a contingency table like the following:

```
> table(grass)
    graze
rich mow unmow
  7    0     1
  8    0     1
  9    0     2
  11   1     0
  12   1     0
  15   2     0
  17   1     0
```

You see the numerical data in the first column, followed by a column for each of the `graze` treatments. The table shows you how many times a particular numerical value cropped up in each of the `graze` treatments.

When your data are all numeric you get a more complex table as a result, because each numeric value in the second column is treated as a separate "level" and compared to each value in the first column. In the following example you have a simple data frame that contains two columns of numeric data:

```
> fw
         count speed
Taw          9     2
Torridge    25     3
Ouse        15     5
Exe          2     9
Lyn         14    14
Brook       25    24
Ditch       24    29
Fal         47    34

> table(fw)
     speed
count 2 3 5 9 14 24 29 34
   2  0 0 0 1  0  0  0  0
   9  1 0 0 0  0  0  0  0
   14 0 0 0 0  1  0  0  0
   15 0 0 1 0  0  0  0  0
   24 0 0 0 0  0  0  1  0
   25 0 1 0 0  0  1  0  0
   47 0 0 0 0  0  0  0  1
```

If you have more complex data frames (that is, with more than two columns), you get a more complex result; you end up with multiple tables. Each of the individual tables shows the first two columns in the data frame, but the values in the other columns are picked out one by one with all the various combinations. This can get complicated to say the least. Here is a simple example where you have a data frame containing a column of numeric values and two columns of factors (character variables):

```
> pw
   height    plant water
1       9 vulgaris    lo
2      11 vulgaris    lo
3       6 vulgaris    lo
4      14 vulgaris   mid
5      17 vulgaris   mid
6      19 vulgaris   mid
7      28 vulgaris    hi
8      31 vulgaris    hi
9      32 vulgaris    hi
10      7   sativa    lo
11      6   sativa    lo
12      5   sativa    lo
13     14   sativa   mid
14     17   sativa   mid
15     15   sativa   mid
16     44   sativa    hi
17     38   sativa    hi
18     37   sativa    hi
```

If you use a `table()` command on these data you get three tables produced as a result. The command produces a table for each of the `water` treatments. Here only the first one is shown:

```
> table(pw)
, , water = hi

      plant
height sativa vulgaris
    5       0        0
    6       0        0
    7       0        0
    9       0        0
    11      0        0
    14      0        0
    15      0        0
    17      0        0
    19      0        0
    28      0        1
    31      0        1
    32      0        1
    37      1        0
    38      1        0
    44      1        0
```

The first table shown examines the situation for the first treatment; the factors are considered in order so the `hi` treatment comes first (because of alphabetical sorting). The other factors are shown in separate tables; the next one being for the `lo` treatment:

```
, , water = lo

      plant
height sativa vulgaris
    5       1        0
    6       1        1
    7       1        0
    9       0        1
...
```

Here just the first few lines are shown of the next table in the set. The final table in the example shows the situation when the `water` treatment is set to `mid`:

```
, , water = mid

      plant
height sativa vulgaris
    5       0        0
    6       0        0
    7       0        0
    9       0        0
...
```

If you have more than a couple of columns of data, things can rapidly get out of hand, and you may end up with a lot more output than you bargained for. You need a way to control which columns are summarized by the table. You can do this with a bit of tweaking to the `table()` command.

Creating Custom Contingency Tables

Rather than use all the columns (or rows) of a data frame, you could create a contingency table that uses only part of the data. In this situation you can actually select each separate row and column to use, as detailed in the following sections.

Selecting Columns to Use in a Contingency Table

The `table()` command enables you to specify which columns of data you want to use to create your contingency table; simply provide the names of the vector objects in the command instruction:

```
> table(height, water)
Error in table(height, water) : object 'height' not found
```

In this example though the command cannot find the objects you want because they are part of the `pw` data frame. You can get around this in one of several ways. You could use `$` and specify the full name or you could use the `attach()` command to "open up" the data frame. In the following example the `$` syntax is used:

```
> table(pw$height, pw$water)

     hi lo mid
  5   0  1   0
```

```
    6   0  2   0
    7   0  1   0
    9   0  1   0
    11  0  1   0
    14  0  0   2
    15  0  0   1
    17  0  0   2
    19  0  0   1
    28  1  0   0
    31  1  0   0
    32  1  0   0
    37  1  0   0
    38  1  0   0
    44  1  0   0
```

The result is a table that shows you a column for each of the three `water` treatments. Notice that the names of the items are not given; the `height` and `water` labels are gone and you only get the names of the three `water` treatments. If you had used the `attach()` command the names would be shown. Of course, there is a way around this; you can use your own labels. To do so, specify the names you require as labels as an instruction in the `table()` command like so:

```
> table(pw$height, pw$water, dnn = c('Ht', 'H2O'))
    H2O
Ht   hi lo mid
  5   0  1   0
  6   0  2   0
  7   0  1   0
...
```

Here you have used a slightly different label than the name of the original vectors; you see the first few lines of the table and can see your customized labels at the top. There is yet another way to get the names of the vectors within the data frame recognized: you can use the `with()` command. This is like using the `attach()` command but you do not have to remember to `detach()` the data afterwards.

The `with()` command works in a general manner. To use it you simply put in the name of the data you want to access and then carry on with the command you wanted to use all in the same line, like so:

```
> with(pw, table(height, water))
```

Now the names of the data columns are available to the `table()` command and the labels reappear:

```
      water
height hi lo mid
    5   0  1   0
    6   0  2   0
    7   0  1   0
...
```

Selecting Rows to Use in a Contingency Table

If you want to use only certain rows of a data frame to form the basis for a contingency table, you need to use a slightly different approach. Essentially this involves creating a matrix object and making a contingency table from that. This topic is discussed in the next section.

Creating Contingency Tables from Matrix Objects

So far you have looked at using the `table()` command on vectors and data frames. You can use the command on matrix objects too, but you need to remember that they have a slightly different structure. Here is a matrix that you have met before; bird observation data:

```
> bird
              Garden Hedgerow Parkland Pasture Woodland
Blackbird         47       10       40       2        2
Chaffinch         19        3        5       0        2
Great Tit         50        0       10       7        0
House Sparrow     46       16        8       4        0
Robin              9        3        0       0        2
Song Thrush        4        0        6       0        0
```

When you use the `table()` command you get the following result:

```
> table(bird)
bird
 0  2  3  4  5  6  7  8  9 10 16 19 40 46 47 50
 9  4  2  2  1  1  1  1  1  2  1  1  1  1  1  1
```

The data in the matrix are treated like a single vector, so your resulting table displays accordingly. The `$` convention does not work with a matrix, so you are therefore also unable to use the `attach()` command. You can, however, use the square brackets to pick out rows and columns to make into your table:

```
> table(bird[,1], bird[,2], dnn = c('Gdn', 'Hedge'))
    Hedge
Gdn  0 3 10 16
  4  1 0  0  0
  9  0 1  0  0
  19 0 1  0  0
  46 0 0  0  1
  47 0 0  1  0
  50 1 0  0  0
```

In this (preceding) example you used the first and second columns and created your contingency table; you need to specify the names for display using the `dnn =` instruction. You can do something similar for row data:

```
> table(bird[3,], bird[1,], dnn = c('Gt. Tit', 'BlackBrd'))
       BlackBrd
Gt. Tit 2 10 40 47
     0  1  1  0  0
     7  1  0  0  0
     10 0  0  1  0
     50 0  0  0  1
```

Here you chose the third and first rows to compare and make up into a contingency table. Once again you must include explicit names for your table display.

With a bit of manipulation you can make your matrix into a data frame and read the column names using the `with()` command:

```
> with(as.data.frame(bird), table(Garden, Pasture))
      Pasture
Garden 0 2 4 7
```

```
    4  1 0 0 0
    9  1 0 0 0
    19 1 0 0 0
    46 0 0 1 0
    47 0 1 0 0
    50 0 0 0 1
```

In this example you use the `as.data.frame()` command to temporarily convert the object to a data frame.

Using Rows of a Data Frame in a Contingency Table

Using a matrix means that you are able to look at the rows to construct your contingency table. If you have a data frame, perhaps you could use the same square bracket convention to do the same thing? Try it in the following example:

```
> fw
         count speed
Taw          9     2
Torridge    25     3
Ouse        15     5
Exe          2     9
Lyn         14    14
Brook       25    24
Ditch       24    29
Fal         47    34

> table(fw[1,], fw[2,])
Error in sort.list(y) :
  'x' must be atomic for 'sort.list'
Have you called 'sort' on a list?
```

The short answer is no, you cannot! You could try the same trick as before and convert the object to a matrix though:

```
> with(as.matrix(fw), table(fw[1,], fw[2,]))
Error in eval(substitute(expr), data, enclos = parent.frame()) :
  numeric 'envir' arg not of length one
```

However, this also fails. The only way to get this to work is to force each item in the table as a matrix like so:

```
> table(as.matrix(fw)[1,], as.matrix(fw)[2,], dnn = c('Taw', 'Torridge'))
   Torridge
Taw 3 25
  2 1  0
  9 0  1
```

You could also make a new matrix and then apply the `table()` command to the new object:

```
> fw.mat = as.matrix(fw)
> table(fw.mat[1,], fw.mat[4,], dnn = c('Taw', 'Exe'))
   Exe
Taw 2 9
  2 0 1
  9 1 0
> rm(fw.mat)
```

Rotating Data Frames

You can rotate the data so that the rows become the columns and vice versa. You can use the `t()` command to transpose a data frame like so:

```
> t(fw)
      Taw Torridge Ouse Exe Lyn Brook Ditch Fal
count   9       25   15   2  14    25    24  47
speed   2        3    5   9  14    24    29  34
```

Now the result is a matrix, so you need to use the square brackets to select the columns (that is, the original rows) that you require:

```
> table(t(fw)[,1], t(fw)[,2], dnn = c('Taw', 'Torridge'))
   Torridge
Taw 3 25
  2 1  0
  9 0  1
```

This approach will also work on a matrix. You can use the `t()` command to rotate a matrix in exactly the same way as for a data frame.

Selecting Parts of a Table Object

A table is a special sort of matrix, and you deal with tables in similar ways to matrix objects. You can extract various elements of a table object exactly as you would for a matrix object. In the following activity you create a contingency table and select out various components of it.

TRY IT OUT Selecting and Displaying Parts of a Contingency Table

Available for download on Wrox.com

Use the `pw` data object from the `Beginning.RData` file for this activity; you will use this to create and examine a custom contingency table.

1. Use the `pw` data frame as the starting point for a custom contingency table:

```
> pw.tab = with(pw, table(height, water))
```

2. View the resulting contingency table:

```
> pw.tab
      water
height hi lo mid
    5   0  1   0
    6   0  2   0
    7   0  1   0
    9   0  1   0
    11  0  1   0
    14  0  0   2
    15  0  0   1
    17  0  0   2
    19  0  0   1
    28  1  0   0
    31  1  0   0
    32  1  0   0
    37  1  0   0
    38  1  0   0
    44  1  0   0
```

3. Examine the table object structure using the `str()` command:

```
> str(pw.tab)
 'table' int [1:15, 1:3] 0 0 0 0 0 0 0 0 0 1 ...
 - attr(*, "dimnames")=List of 2
  ..$ height: chr [1:15] "5" "6" "7" "9" ...
  ..$ water : chr [1:3] "hi" "lo" "mid"
```

4. Now display only the first three rows of the contingency table:

```
> pw.tab[1:3,]
      water
height hi lo mid
     5  0  1   0
     6  0  2   0
     7  0  1   0
```

5. Next display the first three rows of the first column:

```
> pw.tab[1:3,1]
5 6 7
0 0 0
```

6. Now display the first three rows of the first and second columns:

```
> pw.tab[1:3,1:2]
      water
height hi lo
     5  0  1
     6  0  2
     7  0  1
```

7. Display the column labeled `hi`:

```
> pw.tab[,'hi']
 5  6  7  9 11 14 15 17 19 28 31 32 37 38 44
 0  0  0  0  0  0  0  0  0  1  1  1  1  1  1
```

8. Now display the first three rows of two of the columns:

```
> pw.tab[1:3, c('hi', 'mid')]
      water
height hi mid
     5  0   0
     6  0   0
     7  0   0
```

9. Display some of the columns in a new order:

```
> pw.tab[1:3, c('mid', 'hi')]
      water
height mid hi
     5   0  0
     6   0  0
     7   0  0
```

10. Try displaying two columns using a mix of name and number:

```
> pw.tab[,c('hi',3)]
Error: subscript out of bounds
```

11. Look at the length of the table object:

```
> length(pw.tab)
[1] 45
```

12. Finally, display some consecutive items:

```
> pw.tab[16:30]
 [1] 1 2 1 1 1 0 0 0 0 0 0 0 0 0 0
```

How It Works

The first step is to create the table object using two of the columns to produce a simple contingency table. The `str()` command shows that the resulting object is a table. The table can be displayed much like a matrix by using the square brackets to define the rows and columns required. The rows and columns can be specified as numbers or names (if appropriate), but you cannot mix names and numbers in the same command.

The `length()` command produces a result that reflects the number of items in the table; this is similar to a matrix but different from a data frame (where the command produced the number of columns).

A table object is a special kind of object in its own right, but it also has certain properties of a matrix. This will become important to remember when you begin to develop logical tests, as you will see shortly in the section, "Testing for Table Objects."

Converting an Object into a Table

You can convert an object into a table by using the `as.table()` command if it is already a matrix because they are very similar objects. If you have a data frame, however, you must convert it to a matrix first and then convert that into a table. You can do this in one go as follows (however, if you have no row names you may end up with character labels rather than numbers):

```
> as.table(as.matrix(mf))
    len     sp    alg    no3    bod
A  20.00  12.00  40.00   2.25 200.00
B  21.00  14.00  45.00   2.15 180.00
C  22.00  12.00  45.00   1.75 135.00
D  23.00  16.00  80.00   1.95 120.00
E  21.00  20.00  75.00   1.95 110.00
F  20.00  21.00  65.00   2.75 120.00
G  19.00  17.00  65.00   1.85  95.00
...
```

In this case your row names have ended up as uppercase characters; here you can see only the first seven rows of the result. If you try to convert a table directly into an object you get an error, like so:

```
> as.table(mf)
Error in as.table.default(mf) : cannot coerce into a table
```

If you have a list object you have to do a few contortions to get the data in the right form. First, you must use the `stack()` command to get the individual elements out in a frame-like form. Then you can convert to a matrix and finally a table; quite a performance!

In the following example you begin with a simple list containing only two items:

```
> grass.l
$mow
[1] 12 15 17 11 15

$unmow
[1] 8 9 7 9

> gr.tab = as.table(as.matrix(stack(grass.l)))

> colnames(gr.tab) = c('spp', 'graze')

> gr.tab
  spp graze
A 12  mow
B 15  mow
C 17  mow
D 11  mow
E 15  mow
F  8  unmow
G  9  unmow
H  7  unmow
I  9  unmow
```

You have to set the names for the columns separately because the `stack()` command produces default names of `values` and `ind`. Note that you use the `colnames()` command to alter the names.

Testing for Table Objects

You can test to see if an object is a table using the `is.table()` command. This produces a `TRUE` result if you do have a table and a `FALSE` result if you do not:

```
> is.table(bird)
[1] FALSE
> is.table(gr.tab)
[1] TRUE
```

You can also use the `class()` command to see if an object is a table directly:

```
> class(gr.tab)
[1] "table"
```

The `class()` command can form the basis of a logical test by using the `if()` command in the following manner:

```
> if(class(gr.tab) =='table') TRUE else FALSE
[1] TRUE
```

NOTE *The* `if()` *command is useful in enabling options. The basic form of the command is as follows:*

```
if(condition) what.to.do.if.TRUE else what.to.do.if.FALSE
```

You set the `condition` *to test in the main brackets. After this you put what to do if the result of the* `condition` *is* `TRUE`*. The* `else` *part allows you to specify what to do if the result of the condition is* `FALSE`*.*

Complex (Flat) Tables

You can use an alternative version of the `table()` command to make "flat" tables; that is, rather than make several tables, the rows or columns are subdivided to make a single table. The command is `ftable()` and you can use it in several ways.

Making "Flat" Contingency Tables

In the following example you see the plant watering data frame that you met earlier. This has a column of numerical `height` data and two columns of factors, `plant` and `water`. When you create the "flat" contingency table you get something like the following:

```
> ftable(pw)
                water hi lo mid
height plant
5      sativa          0  1   0
       vulgaris        0  0   0
6      sativa          0  1   0
       vulgaris        0  1   0
7      sativa          0  1   0
       vulgaris        0  0   0
9      sativa          0  0   0
       vulgaris        0  1   0
11     sativa          0  0   0
       vulgaris        0  1   0
14     sativa          0  0   1
       vulgaris        0  0   1
15     sativa          0  0   1
       vulgaris        0  0   0
17     sativa          0  0   1
       vulgaris        0  0   1
19     sativa          0  0   0
       vulgaris        0  0   1
28     sativa          0  0   0
       vulgaris        1  0   0
31     sativa          0  0   0
       vulgaris        1  0   0
32     sativa          0  0   0
       vulgaris        1  0   0
37     sativa          1  0   0
       vulgaris        0  0   0
38     sativa          1  0   0
```

```
        vulgaris        0  0   0
44      sativa          1  0   0
        vulgaris        0  0   0
```

Here you can see that the rows are subdivided between the two levels of the `plant` factor.

You can use the command much like the `table()` command and specify two or more columns of the data to use in the table. The order of the columns you specify are used to construct the contingency table. You can also use a slightly different syntax to tell R what sort of output you require; the general form of the command is as follows:

```
ftable(column.items ~ row.items, data = data.object)
```

You use the tilde (~) character to create a formula where the left side contains the variables you want to use as the row headings (in other words, stuff that forms the columns), separated by commas. After the tilde you put the names of the vectors that you want to form the row items.

These commands give you great flexibility in creating contingency tables. In the following activity you practice creating some "flat" contingency tables.

TRY IT OUT Create "Flat" Contingency Tables from Complex Data

Available for download on Wrox.com

Use the `pw` data from the `Beginning.RData` file for this activity, which you will use to create a "flat" contingency table.

1. Start by creating a contingency table from the `pw` data object. Use the columns in the order in which they appear in the original data:

```
> with(pw, ftable(height, plant, water))
                water hi lo mid
height plant
5      sativa          0  1   0
       vulgaris        0  0   0
6      sativa          0  1   0
       vulgaris        0  1   0
7      sativa          0  1   0
       vulgaris        0  0   0
9      sativa          0  0   0
       vulgaris        0  1   0
...
```

2. Now create another contingency table but specify the columns in a new order:

```
> with(pw, ftable(height,water,plant))
              plant sativa vulgaris
height water
5      hi                0        0
       lo                1        0
       mid               0        0
6      hi                0        0
       lo                1        1
       mid               0        0
...
```

3. Next try creating a flat table using the ~ syntax. Keep the same column order as the first table you created:

```
> ftable(plant ~ height + water, data = pw)
             plant sativa vulgaris
height water
5      hi              0        0
       lo              1        0
       mid             0        0
6      hi              0        0
       lo              1        1
       mid             0        0
...
```

4. Now try to create the same table as the first but using the new ~ syntax:

```
> ftable(water ~ height + plant, data = pw)
                water hi lo mid
height plant
5      sativa          0  1   0
       vulgaris        0  0   0
6      sativa          0  1   0
       vulgaris        0  1   0
...
```

5. Now specify the main response variable as the main grouping variable in your flat table:

```
> ftable(height ~ water + plant, data = pw)
               height 5 6 7 9 11 14 15 17 19 28 31 32 37 38 44
water plant
hi    sativa          0 0 0 0  0  0  0  0  0  0  0  0  1  1  1
      vulgaris        0 0 0 0  0  0  0  0  0  1  1  1  0  0  0
lo    sativa          1 1 1 0  0  0  0  0  0  0  0  0  0  0  0
      vulgaris        0 1 0 1  1  0  0  0  0  0  0  0  0  0  0
mid   sativa          0 0 0 0  0  1  1  1  0  0  0  0  0  0  0
      vulgaris        0 0 0 0  0  1  0  1  1  0  0  0  0  0  0
```

6. Finally, re-create the last table without the ~ syntax:

```
> with(pw, ftable(water, plant, height))
```

How It Works

To start with you had to use the `with()` command so that the names of the columns in the original data could be "read" by R. The order in which you enter the columns is very important; in this case `height` was specified first and this forms the column margin of the table. The next item inserted was `plant` followed by `water`; is the order in which they appear. If you change the order, obviously the appearance of the table is different and your data are summarized in a different way that may make more (or less) sense. In the next table `height` was kept as the main column, but the order of the factor variables was swapped.

The ~ syntax allows great flexibility and enables you to create contingency tables relatively easily. The column specified before the ~ forms the main body of the table, whereas those to the right of the ~ form the groupings of the table in the order they were specified. The final example shows that the most compact table uses the main response variable as the main body.

You can extract parts of your table exactly using the square brackets, like you did in an earlier activity. In the next example the contingency table contains two data columns; these relate to the different levels in the `plant` column:

```
> gr.t = ftable(plant ~ height + water, data = pw)

> gr.t
             plant sativa vulgaris
height water
5      hi               0        0
       lo               1        0
       mid              0        0
6      hi               0        0
       lo               1        1
       mid              0        0
...
```

You can now use the square brackets to extract rows and columns of the resulting contingency table object.

```
> gr.t[1:3,]
     [,1] [,2]
[1,]    0    0
[2,]    1    0
[3,]    0    0
```

Here only the first three rows are selected. Notice that the names are not displayed. The simplest way to extract a part of the object is to make a logical unit of some sort and to create a separate matrix from it and name that. For example, you can use the first three rows from the preceding example; these relate to the `height` of 5. Make a matrix and give it sensible names before displaying the full result:

```
> gr.sub = gr.t[1:3,]
> gr.sub
     [,1] [,2]
[1,]    0    0
[2,]    1    0
[3,]    0    0

> colnames(gr.sub) = c('sativa', 'vulgaris')
> rownames(gr.sub) = c('hi','lo','mid')

> gr.sub
    sativa vulgaris
hi       0        0
lo       1        0
mid      0        0
```

You could also do this in one go by specifying the names as part of the command like so:

```
> gr.sub = matrix(gr.t[1:3,],ncol =2, dimnames = list(c('hi','lo','mid'),
c('sativa','vulgaris')))
```

This time you have to use the `matrix()` command to set the row and column names (using the `dimnames` instruction).

Making Selective "Flat" Contingency Tables

A contingency table is best kept as a complete item. As you saw earlier, you can subset a "flat" table using the square brackets syntax. However, this is not a straightforward process. It would be better if you could create a contingency table right at the outset that matched certain conditions. The following activity shows how you can do this.

TRY IT OUT Creating Selective "Flat" Contingency Tables

Available for download on Wrox.com

Use the `pw` data object from the `Beginning.RData` file for this activity, which you will use to create a "flat" contingency table.

1. Start by creating a "flat" contingency table with a conditional column:

```
> with(pw, ftable(height==14, water, plant))
           plant sativa vulgaris
     water
FALSE hi              3        3
      lo              3        3
      mid             2        2
TRUE  hi              0        0
      lo              0        0
      mid             1        1
```

2. Now add an additional condition to another column:

```
> with(pw, ftable(height==14, water=='hi', plant))
           plant sativa vulgaris

FALSE FALSE           5        5
      TRUE            3        3
TRUE  FALSE           1        1
      TRUE            0        0
```

3. Make a new data object as a subset of the original data:

```
> pw.t = pw[which(pw$height==14),]

> pw.t
   height    plant water
4      14 vulgaris   mid
13     14   sativa   mid
```

4. Finally, create a "flat" table from the new (subsetted) data:

```
> with(pw.t, ftable(height, plant, water))
               water hi lo mid
height plant
14     sativa         0  0   1
       vulgaris       0  0   1
```

How It Works

When you insert a conditional column into the `ftable()` command, the resulting contingency table includes the data for both `TRUE` and `FALSE` results of the condition. You can add conditional statements for other columns (and also more complex conditional statements for the single column) and produce more `TRUE` and `FALSE` results.

To make a selective ftable object you must create a new data frame that contains only the data you require. Now you can use the ftable() command on the new data to produce a result that contains no TRUE or FALSE results, only "real" data.

Now you have seen how to create "flat" contingency tables and how to be selective when doing so, using only certain data and even including conditional statements to create a selective table. You now need to be able to tell if an object is a "flat" table; that is the subject of the next section.

Testing "Flat" Table Objects

You can use the class(), command to see what kind of object you are dealing with. The class() command gives you a label for each kind of object. The class of an object is used to determine how R handles it; you can find out what an object is and also set the class of an object. The bottom line is that you can use the class to test if your object is an ftable object by looking at its class like so:

```
> if(class(gr.t) == 'ftable') TRUE else FALSE
[1] TRUE
```

Here you look to see if the class is "ftable"; if it is, you give a TRUE result, otherwise you get a FALSE result.

Summary Commands for Tables

A table is usually a way of summarizing some data and is often the end point of an operation (for example, making a contingency table). At times, however, you may want to perform certain actions on a table itself. You have already met some commands that could be useful, for example rowMeans(), colSums(), and apply(). These will work equally well on a table as they will on a matrix. However, you can use some additional commands, which are summarized in Table 4-5.

TABLE 4-5: Table Summary Commands

SUMMARY COMMAND	EXPLANATION

rowSums()
colSums()
	Determines the sum of rows or columns for a data frame, matrix, or table object.

rowMeans()
colMeans()
	Determines the mean of rows or columns for a data frame, matrix, or table object.

apply(x, MARGIN, FUN)	Applies a function to rows or columns of a data frame, matrix, or table. If MARGIN = 1 the rows are used, if 2 the columns are used.
prop.table(x, margin = NULL, FUN)	Returns the contents of a data frame, matrix, or table as a proportion of the total specified margin. The default uses the grand total, margin = 1 uses row totals, and margin = 2 uses column totals.
addmargins(A, margin = c(1, 2), FUN = sum)	Returns a function applied to rows and/or columns of a matrix or table.

In the following activity you try out some of these summary commands to get a feeling for what they can do for you. You use a table of bird observation data that you have met before. The object in question is actually a matrix object but, as you have seen, a matrix acts very much like a table. In this case the way the data are arranged forms a contingency table. Each cell in the table is a unique combination of two factors.

TRY IT OUT Carrying Out Summary Commands on a Contingency Table

Available for download on Wrox.com

Use the `bird` data from the `Beginning.RData` file for this activity, which you will use as the contingency table to explore.

1. Start by looking at the `bird` data object:

```
> bird
             Garden Hedgerow Parkland Pasture Woodland
Blackbird        47       10       40       2        2
Chaffinch        19        3        5       0        2
Great Tit        50        0       10       7        0
House Sparrow    46       16        8       4        0
Robin             9        3        0       0        2
Song Thrush       4        0        6       0        0
```

2. Use the `rowSums()` command to look at sums of rows:

```
> rowSums(bird)
    Blackbird     Chaffinch     Great Tit House Sparrow         Robin   Song Thrush
          101            29            67            74            14            10
```

3. Now try the `apply()` command to look at column sums:

```
> apply(bird, MARGIN = 2, FUN = sum)
  Garden Hedgerow Parkland  Pasture Woodland
     175       32       69       13        6
```

4. Use the `margin.table()` command to get an overall total:

```
> margin.table(bird)
[1] 295
```

5. Use the `margin.table()` command to determine row sums:

```
> margin.table(bird, 1)
    Blackbird     Chaffinch     Great Tit House Sparrow       Robin    Song Thrush
          101            29            67            74          14             10
```

6. Now use the `margin.table()` command to determine column sums:

```
> margin.table(bird, margin = 2)
  Garden Hedgerow Parkland  Pasture Woodland
     175       32       69       13        6
```

7. Use the `prop.table()` command to display the table data as proportions of the total sum:

```
> prop.table(bird)
                  Garden   Hedgerow   Parkland     Pasture    Woodland
Blackbird     0.15932203 0.03389831 0.13559322 0.006779661 0.006779661
Chaffinch     0.06440678 0.01016949 0.01694915 0.000000000 0.006779661
```

```
Great Tit     0.16949153 0.00000000 0.03389831 0.023728814 0.000000000
House Sparrow 0.15593220 0.05423729 0.02711864 0.013559322 0.000000000
Robin         0.03050847 0.01016949 0.00000000 0.000000000 0.006779661
Song Thrush   0.01355932 0.00000000 0.02033898 0.000000000 0.000000000
```

8. Add a margin instruction to the `prop.table()` command to display the table as proportions of the row totals:

```
> prop.table(bird, margin = 1)
                 Garden  Hedgerow  Parkland    Pasture   Woodland
Blackbird     0.4653465 0.0990099 0.3960396 0.01980198 0.01980198
Chaffinch     0.6551724 0.1034483 0.1724138 0.00000000 0.06896552
Great Tit     0.7462687 0.0000000 0.1492537 0.10447761 0.00000000
House Sparrow 0.6216216 0.2162162 0.1081081 0.05405405 0.00000000
Robin         0.6428571 0.2142857 0.0000000 0.00000000 0.14285714
Song Thrush   0.4000000 0.0000000 0.6000000 0.00000000 0.00000000
```

9. Now use the `addmargins()` command to determine a row of mean values for the table:

```
> addmargins(bird, 1, mean)
                Garden  Hedgerow Parkland  Pasture Woodland
Blackbird     47.00000 10.000000     40.0 2.000000        2
Chaffinch     19.00000  3.000000      5.0 0.000000        2
Great Tit     50.00000  0.000000     10.0 7.000000        0
House Sparrow 46.00000 16.000000      8.0 4.000000        0
Robin          9.00000  3.000000      0.0 0.000000        2
Song Thrush    4.00000  0.000000      6.0 0.000000        0
mean          29.16667  5.333333     11.5 2.166667        1
```

10. Use the `addmargins()` command to work out a column median for the table:

```
> addmargins(bird, 2, median)
              Garden Hedgerow Parkland Pasture Woodland median
Blackbird         47       10       40       2        2     10
Chaffinch         19        3        5       0        2      3
Great Tit         50        0       10       7        0      7
House Sparrow     46       16        8       4        0      8
Robin              9        3        0       0        2      2
Song Thrush        4        0        6       0        0      0
```

How It Works

The `rowSums()` and `colMeans()` commands are general; you have used these before to work out sums and means for rows and columns. The `apply()` command is more flexible; you use it to apply a function to rows (`MARGIN = 1`) or columns (`MARGIN = 2`).

The `margin.table()` command is essentially the same as `apply()` when used with `FUN = sum`. If you leave out the `margin` instruction you get the complete total. Using `margin = 1` gives the row sums and `margin = 2` returns the column sums.

You can use the `prop.table()` command to display the table data as proportions of the total sum. You can add an index for the rows or columns in the same way as for the `margin.table()` command; in this way you can express the data in your table as proportions of the various row or column sums.

The `addmargins()` command enables you to use any function on rows or columns. The margin part defaults to both rows and columns, whereas the function applied defaults to the sum. In this case, the row/column index works like so: A value of 1 refers to rows, but the function is applied to the row items. Essentially, you get a row of results. In most situations you are going to use the function to produce summaries for both rows and columns.

You can see that you have several ways to achieve the same result. In some cases one command is a simpler version of something more complex but in others subtle differences exist.

Cross Tabulation

The table-creating commands that you have looked at so far build up frequencies of observations across categories. However, you may already have the frequency data, and in this case you need to reassemble the data into a tabular format. You met one table that had already been created when you looked at the bird observation data. If you look at the raw data file, you see that there are three columns: one for the species, one for the habitats, and one for the quantity (that is, the frequency of observations):

```
> birds
         Species  Habitat Qty
1      Blackbird   Garden  47
2      Chaffinch   Garden  19
3      Great Tit   Garden  50
4  House Sparrow   Garden  46
5          Robin   Garden   9
6    Song Thrush   Garden   4
7      Blackbird Parkland  40
8      Chaffinch Parkland   5
9      Great Tit Parkland  10
10 House Sparrow Parkland   8
11   Song Thrush Parkland   6
12     Blackbird Hedgerow  10
13     Chaffinch Hedgerow   3
14 House Sparrow Hedgerow  16
15         Robin Hedgerow   3
16     Blackbird Woodland   2
17     Chaffinch Woodland   2
18         Robin Woodland   2
19     Blackbird Pasture    2
20     Great Tit Pasture    7
21 House Sparrow Pasture   4
```

These data are in data frame format and were read into R using the `read.csv()` command. Your task is to reorganize the data into a contingency table; if you try this using the `table()` command you end up with a simple table like so:

```
> with(birds, table(Species, Habitat))
               Habitat
Species          Garden Hedgerow Parkland Pasture Woodland
  Blackbird           1        1        1       1        1
  Chaffinch           1        1        1       0        1
  Great Tit           1        0        1       1        0
  House Sparrow       1        1        1       1        0
  Robin               1        1        0       0        1
  Song Thrush         1        0        1       0        0
```

This is not really what you want because this shows you only which categories have observations (your data are reduced to ones and zeros). If you add the `Qty` column into the mix you end up with multiple tables, one for each `Qty`:

```
> with(birds, table(Species,Habitat,Qty))
, , Qty = 2

               Habitat
Species          Garden Hedgerow Parkland Pasture Woodland
  Blackbird           0        0        0       1        1
  Chaffinch           0        0        0       0        1
  Great Tit           0        0        0       0        0
  House Sparrow       0        0        0       0        0
  Robin               0        0        0       0        1
  Song Thrush         0        0        0       0        0

, , Qty = 3

               Habitat
Species          Garden Hedgerow Parkland Pasture Woodland
  Blackbird           0        0        0       0        0
  Chaffinch           0        1        0       0        0
  Great Tit           0        0        0       0        0
  House Sparrow       0        0        0       0        0
  Robin               0        1        0       0        0
  Song Thrush         0        0        0       0        0
...
```

Here just the first couple of tables produced are shown. The `ftable()` command gives you a single table, but you still end up with 1s and 0s:

```
> with(birds, ftable(Species,Habitat, Qty))
                          Qty 2  3  4  5  6  7  8  9 10 16 19 40 46 47 50
Species       Habitat
Blackbird     Garden          0  0  0  0  0  0  0  0  0  0  0  0  0  1  0
              Hedgerow        0  0  0  0  0  0  0  0  1  0  0  0  0  0  0
              Parkland        0  0  0  0  0  0  0  0  0  0  0  1  0  0  0
              Pasture         1  0  0  0  0  0  0  0  0  0  0  0  0  0  0
              Woodland        1  0  0  0  0  0  0  0  0  0  0  0  0  0  0
Chaffinch     Garden          0  0  0  0  0  0  0  0  0  0  1  0  0  0  0
              Hedgerow        0  1  0  0  0  0  0  0  0  0  0  0  0  0  0
              Parkland        0  0  0  1  0  0  0  0  0  0  0  0  0  0  0
              Pasture         0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
              Woodland        1  0  0  0  0  0  0  0  0  0  0  0  0  0  0
...
```

The first few lines of the resulting table are shown here. You want to end up with the original frequency data (titled `Qty` in the data frame). To do that you use a cross-tabulation command called `xtabs()`. The basic form of the command is as follows:

```
xtabs(freq.data ~ categories.list, data)
```

Notice that you have the tilde (~) symbol like you met when using the `ftable()` command. On the left of the ~ you put the name of the frequency data; on the right you put the categories you want to cross-tabulate separated by the plus sign. The first variable after the ~ forms the row categories and the next variable you type forms the columns categories. At the end you type the name of the data object (so that R can "find" the variables). For the bird observation data you would type something like the following:

```
> birds.t = xtabs(Qty ~ Species + Habitat, data = birds)
> birds.t
                Habitat
Species          Garden Hedgerow Parkland Pasture Woodland
  Blackbird          47       10       40       2        2
  Chaffinch          19        3        5       0        2
  Great Tit          50        0       10       7        0
  House Sparrow      46       16        8       4        0
  Robin               9        3        0       0        2
  Song Thrush         4        0        6       0        0
```

This does the job nicely and now you see the data rearranged as required.

Testing Cross-Table (xtabs) Objects

When you use the `xtabs()` command, the object you create is a kind of table and gives a `TRUE` result using the `is.table()` command. It also gives a `TRUE` result if you use the `as.matrix()` command. As far as R is concerned it holds two sorts of `class`. You can see this using the `class()` command:

```
> class(birds.t)
[1] "xtabs" "table"
```

If you want to test for the object being an `xtabs` object you have a problem, because now the class result has two elements, shown in the following:

```
> if(class(birds.t) == 'xtabs') TRUE else FALSE
[1] TRUE
Warning message:
In if (class(birds.t) == "xtabs") TRUE else FALSE :
  the condition has length > 1 and only the first element will be used
```

When you try it you get a result of sorts but you also get an error message. You are lucky to get a result simply because the `"xtabs"` result was the first. In reality, you need a way of scanning the entire "result" and picking out the bit you want, wherever it may be. You can do that with the following:

```
> if(any(class(birds.t) == 'xtabs')) TRUE else FALSE
[1] TRUE
```

The `any()` command enables you to match any of the elements in a vector. In this case the upshot is that you will pick out the `xtabs` item even if it is not the first in the bunch.

A Better Class Test

The commands for testing for object types that you have seen so far are not the only ones at your disposal. Two other commands are especially useful:

- ➤ `is(object, "type")`
- ➤ `inherits(object, "type")`

Both these commands return a `TRUE` result if the `class()` of an object matches the “type” you specify in the instruction. These are especially powerful and useful because they mean that you do not need to use complicated `if()` and `any()` commands!

Recreating Original Data from a Contingency Table

If you have an `xtabs` object, you can reassemble it into a data frame using the `as.data.frame()` command:

```
> as.data.frame(birds.t)
         Species  Habitat Freq
1      Blackbird   Garden   47
2      Chaffinch   Garden   19
3      Great Tit   Garden   50
4  House Sparrow   Garden   46
5          Robin   Garden    9
6    Song Thrush   Garden    4
7      Blackbird Hedgerow   10
8      Chaffinch Hedgerow    3
9      Great Tit Hedgerow    0
10 House Sparrow Hedgerow   16
11         Robin Hedgerow    3
12   Song Thrush Hedgerow    0
13     Blackbird Parkland   40
14     Chaffinch Parkland    5
15     Great Tit Parkland   10
16 House Sparrow Parkland    8
17         Robin Parkland    0
18   Song Thrush Parkland    6
19     Blackbird Pasture     2
20     Chaffinch Pasture     0
21     Great Tit Pasture     7
22 House Sparrow Pasture     4
23         Robin Pasture     0
24   Song Thrush Pasture     0
25     Blackbird Woodland    2
26     Chaffinch Woodland    2
27     Great Tit Woodland    0
28 House Sparrow Woodland    0
29         Robin Woodland    2
30   Song Thrush Woodland    0
```

Now you have re-created the original data with one or two minor differences. The `Qty` column has been renamed `Freq` and the rows with zero frequency are included. Neither of these are significant issues; you can alter the name of the `Freq` column by amending the command like so:

```
> as.data.frame(birds.t, responseName = 'Qty')
```

If you want to remove the zero data you need to take your new data frame and select those rows with a `Freq` greater than zero, like so:

```
> birds.td = as.data.frame(birds.t)
> birds.td = birds.td[which(birds.td$Freq > 0),]
```

You begin by creating a new object to accept the data frame; the object is called `birds.td` in this example. Then you select all rows that have `Freq` greater than `0`. Notice that you have overwritten the original data frame with the amended one in this example. This is not essential but it is done here so that you do not have to delete a temporary object later. If you make a mistake you can easily recall the previous command and start over again.

If you have a contingency table that is not a table object but something like a matrix, then you need to alter the class of the object before you convert the object to a data frame. This is the subject of the next section.

Switching Class

You can use the `class()` command to alter which class an object is as well as see what class the object currently is. This can be useful for occasions where an object needs to be in a certain class for a command to operate. In the following example you can see the `bird` object queried and then reset using the `class()` command:

```
> class(bird)
[1] "matrix"
> class(bird) = 'table'
```

The matrix of bird observations is now classed as a table. You can now proceed to create a data frame from the table using the `as.data.frame()` command:

```
> bird.df = as.data.frame(bird)
            Var1     Var2 Freq
1      Blackbird   Garden   47
2      Chaffinch   Garden   19
3      Great Tit   Garden   50
4  House Sparrow   Garden   46
5          Robin   Garden    9
6    Song Thrush   Garden    4
7      Blackbird Hedgerow   10
8      Chaffinch Hedgerow    3
9      Great Tit Hedgerow    0
10 House Sparrow Hedgerow   16
...
```

Notice that the columns are not labeled appropriately and that the zero data are still intact. You can alter the names of the columns using the `names()` command and reconstruct the data omitting the zero rows as you saw in the preceding section. Here is the entire process:

```
> bird.tt = bird
> class(bird.tt) = 'table'
> bird.tt = as.data.frame(bird.tt)
> names(bird.tt) = c('Species', 'Habitat', 'Qty')
> bird.tt = bird.tt[which(bird.tt$Qty > 0),]
> rownames(bird.tt) = as.numeric(1:length(rownames(bird.tt)))
```

In the first command you simply create a duplicate matrix to work on, keeping your original intact. The second command changes the class to “table”. The third command creates the data frame of original values. The fourth command alters the names of the columns. The penultimate command selects out the data that are greater than zero, effectively deleting 0 observations. The final command reinstates the row index labels to a continuous sequence.

SUMMARY

- You can summarize data items using the `summary()` command, which may give a specific result or a general summary.
- Specific summary commands include `mean()`, `median()`, `max()`, `min()`, `sd()`, `quantile()`, and `length()`.
- Cumulative statistics can be obtained via the `cumsum()` and `cummax()` commands. These can form the basis of simple custom functions to calculate for (for example, the running mean).
- Data frame and matrix objects can have summary functions applied to the rows and columns (for example, `colSums()`, `colMeans()`, `rowSums()`, and `rowMeans()` commands). The `apply()` command allows any function to be applied to rows or columns. The `lapply()` and `sapply()` commands are special variants designed to work on list objects.
- Contingency tables can be made using the `table()` command; the `ftable()` command creates "flat" tables for use with more complex data.
- You can convert raw data into a contingency table using the `xtabs()` command. This cross tabulation is similar to the PivotTable of Excel.
- The `class()` command can be used to tell what kind of object you are dealing with and can form the basis of a logical test.

EXERCISES

Available for download on Wrox.com

You can find answers to the exercises in Appendix A.

Use the `Beginning.RData` file for these exercises; the file contains the required data objects.

1. Have a look at the `mf` data object. Determine what kind of object this is and carry out some simple statistical summaries on these data.
2. Look at the `bfs` data object. Construct contingency tables using both the `table()` and `ftable()` commands. How can you get one command to produce the same layout of table as the other and what is the key difference between these results?
3. Look at the `invert` data object. Here you can see a data frame with three columns. Use cross tabulation to construct a contingency table showing the relationship between `Taxa` and `Habitat`. Save the resulting table as an object. What kind of object do you have and how can you reconstruct the original data?

WHAT YOU LEARNED IN THIS CHAPTER

TOPIC	KEY POINTS
Summarizing objects: `summary()`	The `summary()` command is a general command that provides a summary of an object. If you have numerical data, then you get a numerical summary (for example, mean, max, min) but if the data are text, you get a note of how many different items you have. Using the `summary()` command on the result of many analytical routines produces a special summary suited to the kind of analysis performed.

Summarizing samples:
`mean()`
`median()`
`max()`
`min()`
`sd()`
`var()`
`length()`
`sum()`
`quantile()`
`fivenum()`
	Numerical samples can be summarized by many commands. Simple commands like `mean()` produce a single result (the mean), whereas others produce several. The `quantile()` command, for example, produces five values as its result (the five basic quartiles).

Cumulative statistics:
`cumsum()`
`cummax()`
`cummin()`
`cumproduct()`
`seq_along()`
	Some commands produce cumulative values, for example the `cumsum()` command results in the cumulative sum of a numeric sample. The `seq_along()` command creates a simple index.
	These commands can be combined and used to create a range of cumulative statistics. For example the running mean:
	`cumsum(my.data) / seq(along = my.data)`

Summarizing rows and columns:
`colSums()`
`colMeans()`
`rowSums()`
`rowMeans()`
`apply()`
`lapply()`
`sapply()`
	The rows and columns of two-dimensional data objects can be summarized in various ways. The `colSums()` and `rowMeans()` commands, for example, produce the sum and mean values for columns and rows, respectively.
	The `apply()` command is more f exible in that any function can be applied to the columns (default) or rows of a data frame or matrix. The `lapply()` and `sapply()` commands are similar but are designed to work with list objects.

TOPIC	KEY POINTS

Contingency tables and cross tabulation:
`table()`
`ftable()`
`xtabs()`
	Contingency tables can be created using the `table()` command. When the data contains several columns, a "f at" table can be produced using the `ftable()` command.
	Data can be cross-tabulated to form contingency tables using the `xtabs()` command.
	Contingency tables can be reorganized into a data frame using the `as.data.frame()` command.

Table summaries:
`margin.table()`
`prop.table()`
`addmargins()`
	Tables can be summarized in exactly the same way as data frames and matrix objects by using `apply()`, for example. In addition, several commands are aimed explicitly at contingency tables. The `margin.table()` command gives sums for rows/columns. The `prop.table()` command determines the proportion that table entries make toward the total. The `addmargins()` command applies any function to rows/columns of a table.

Testing table objects:
`is.table()`
`is.matrix()`
`class()`
`any()`
`is(object, "type")`
`inherits(object, "type")`
	You can test to see if an object is of a certain type; using `is.table()` and `is.matrix()` commands, for example, will test for a table and a matrix, respectively. These commands produce a `TRUE` or `FALSE` result.
	The `class()` command can be used to view or set the current type of an object. Objects can have more than one class so if a test is required the `any()` command can be used to match any of the classes that may be present.
	The `is()` and `inherits()` commands can extract the class of an object directly and return a `TRUE` result if the class matches the `"type"`.

Logic and testing:
`for()`
`if() else`
`any()`
	The `for()` command can be used to create loops (for example, in creating cumulative statistics like a running median).
	The `if()` command is used to test some condition and carry out a command if the result is `TRUE`. It can add the command `else` to the end to carry out a command when the result is `FALSE`.
	The `any()` command can be used in testing conditions to match any item in a list.

continues

WHAT YOU LEARNED IN THIS CHAPTER *(continued)*

TOPIC	KEY POINTS

Programming/custom functions:
`any()`
`for()`
`if() else`
`function()`
	The `any()` command enables the matching of any element in a list containing several items.
	The `for()` command is used to create programming loops.
	The `if()` command is used in logical testing of some condition and can be paired with `else` to provide an alternative.
	Customized commands can be created using the `function()` command.

Creating sequences: `seq()`	The `seq()` command produces sequences of values.

Reading data objects:
`attach()`
`detach()`
`with()`
	Variables contained within other data objects, such as the columns of a data frame, are usually inaccessible to R. They can be accessed using the `$` syntax (for example, `my.data$column`).
	Alternatively, the enclosing object can be "opened" using the `attach()` command. The `detach()` command closes the enclosing object.
	The `with()` command enables temporary access to an enclosing object.