Chapter 5

Data:Distribution

What you will learn in this chapter:

- How to create histograms and other graphics of sample distribution How to
- examine various distributions
- How to test for the normal distribution How to
- generate random numbers

Whenever you have data you should strive to find a shorthand way of expressing it. In the previous chapter you looked at summary statistics and tabulation. Visualizing your data is also important, as it is often easier to interpret a graph than a series of numbers. Whenever you have a set of numerical values you should also look to see what the distribution of the data is. The classic normal distribution for example, is only one kind of distribution that your data may appear in.

The distribution is important because most statistical approaches require the data to be in one form. Knowing the distribution of your data will help you towards the correct analytical procedure. This chapter looks at ways to display the distribution of your data in graphical form and at different data distributions. You will also look at ways to test if your data conform to the normal distribution, which is most important for statistical testing. You will also look at random numbers and ways of sampling randomly from within a dataset.

Looking at the Distribution ofData

When doing statistical analysis it is important to get a "picture" of the data. You usually want to know if the observations are clustered around some middle point (the average) and if there are observations way out on their own (outliers). This is all related to the *distribution* of the data. There are many distributions, but common ones are the *normal* distribution, *Poisson*, and *binomial*. There are also distributions relating directly to statistical tests; for example, *chi-squared* and *Student'st*.

It is necessary to look at your data and be able to determine what kind of distribution is most adequately represented by them. It is also useful to be able to compare the distribution you have with one of the standard distributions to see how your sample matchesup.

You already met the `table()` command, which was a step toward gaining an insight into the distribution of a sample. It enables you to see how many observations there are in a range of categories. This section covers other general methods of looking at data and distributions.

Stem and Leaf Plot

The `table()` command gives you a quick look at a vector of data, but the result is still largely numeric. A graphical summary might be more useful because a picture is often easier to interpret. You could draw a frequency histogram, and indeed you do this in the following section, but you can also use a stem and leaf plot as a kind of halfway house. The `stem()` command

produces the stem and leafplot.

In the following activity you will use the `stem()` command and compare it to the `table()` command that you used in Chapter 4.

The `stem()` command is a quick way of assessing the distribution of a sample and is also useful because the original values are shown, allowing the sample to be reconstructed from the result. However, when you have a large sample you end up with a lot of values and the command cannot display the data very well; in such cases a histogram is more useful.

Histograms

The histogram is the classic way of viewing the distribution of a sample. You can create histograms using the graphical command `hist()`, which operates on numerical vectors of data like so:

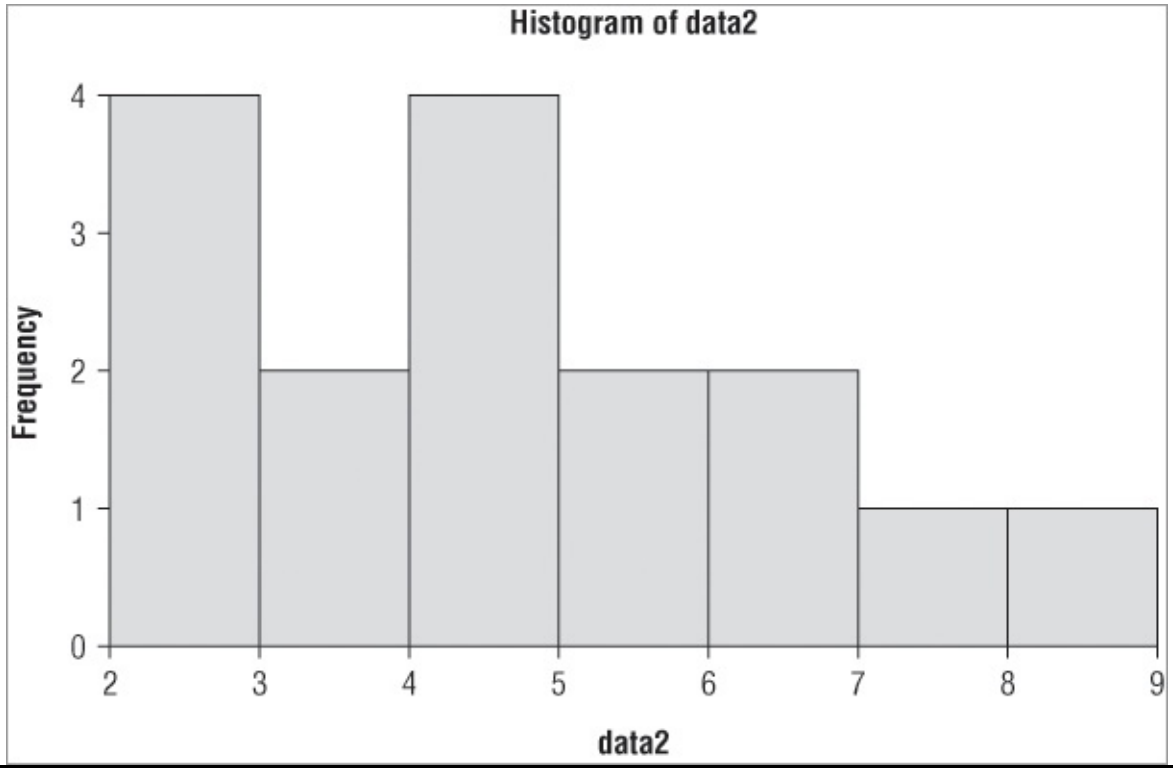
```
> data2
 [1] 3 5 7 5 3 2 6 8 5 6 9 4 5 7 3 4
> hist(data2)
```

In this example you used a simple vector of numerical values (these are integer data) and the resulting histogram looks like Figure 5-1:

the frequencies are represented on the y-axis and the x-axis shows the values separated into various bins. If you use the `table()` command you can see how the histogram is constructed:

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

Figure 5-1



The first bar straddles the range 2 to 3 (that is, greater than 2 but less than 3), and therefore you should expect four items in this bin. The next bar straddles the 3 to 4 range, and if you look at the table you see there are two items bigger than 3 but not bigger than 4. You can alter the number of columns that are displayed using the `breaks =` instruction as part of the command. This instruction will accept several sorts of input; you can use a standard algorithm for calculating the breakpoints, for example. The default is `breaks = "Sturges"`, which uses the range of the data to split into bins. Two other standard algorithms are used: `"Scott"` and `"Freedman-Diaconis"`. You can use lowercase and unambiguous abbreviation; additionally you can use `"FD"` for the last of these three options:

```
> hist(data2, breaks =
'Sturges')
> hist(data2, breaks = 'Scott')
> hist(data2, breaks = 'FD')
```

Thus you might also use the following:

```
> hist(data2, breaks = 'st')
> hist(data2, breaks = 'sc')
> hist(data2, breaks = 'fr')
```

You can also specify the number of breaks as a simple number or range of numbers; the following commands all produce the same result, which for these data is the same as the default (`Sturges`):

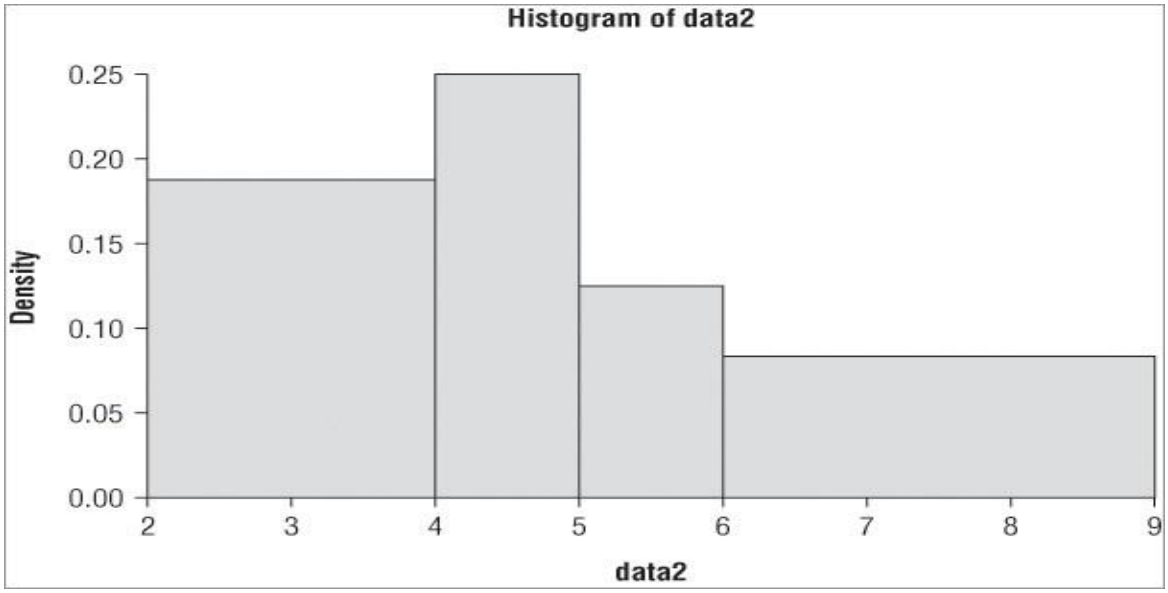
```
> hist(data2, breaks = 7)
> hist(data2, breaks = 2:9)
> hist(data2, breaks = c(2,3,4,5,6,7,8,9))
```

Being able to specify the breaks exactly means that you can produce a histogram with unequal binranges:

```
> hist(data2, breaks = c(2,4,5,6,9))
```

The resulting histogram appears like Figure 5-2.

Figure 5-2



Notice in Figure 5-2 that the y-axis does not show the frequency but instead shows the density. The command has attempted to keep the areas of the bars correct and in proportion (in other words, the total area sums to 1). You can use the `freq =` instruction to produce either frequency data (`TRUE` is the default) or density data(`FALSE`).

You can apply a variety of additional instructions to the `hist()` command. Many of these extra instructions apply to other graphical commands in R (see Table 5-1 for a summary).

Table 5-1: A Summary of Graphical Commands Useful with Histograms

Graphical	Explanation
`col = 'color'`	Thecolorofthebars;acolornameinquotes(use`colors()`toseetherange available).
`main = 'main title'`	A main title for the histogram; use `NULL`to suppress this.
`xlab =`	A title for the x-axis.
`ylab =`	A title for the y-axis.
`xlim = c(start,`	The range for the x-axis; put numerical values for the start and end points.
`ylim = c(start, end)`	The range for the y-axis.

These commands are available for many of the other graphs that you meet using R. The color of the bars is set using the `col =` instruction. A lot of colors are available, and you can see the standard options by using the `colors()` command:

```
colors()
```

You can also use an alternative spelling.

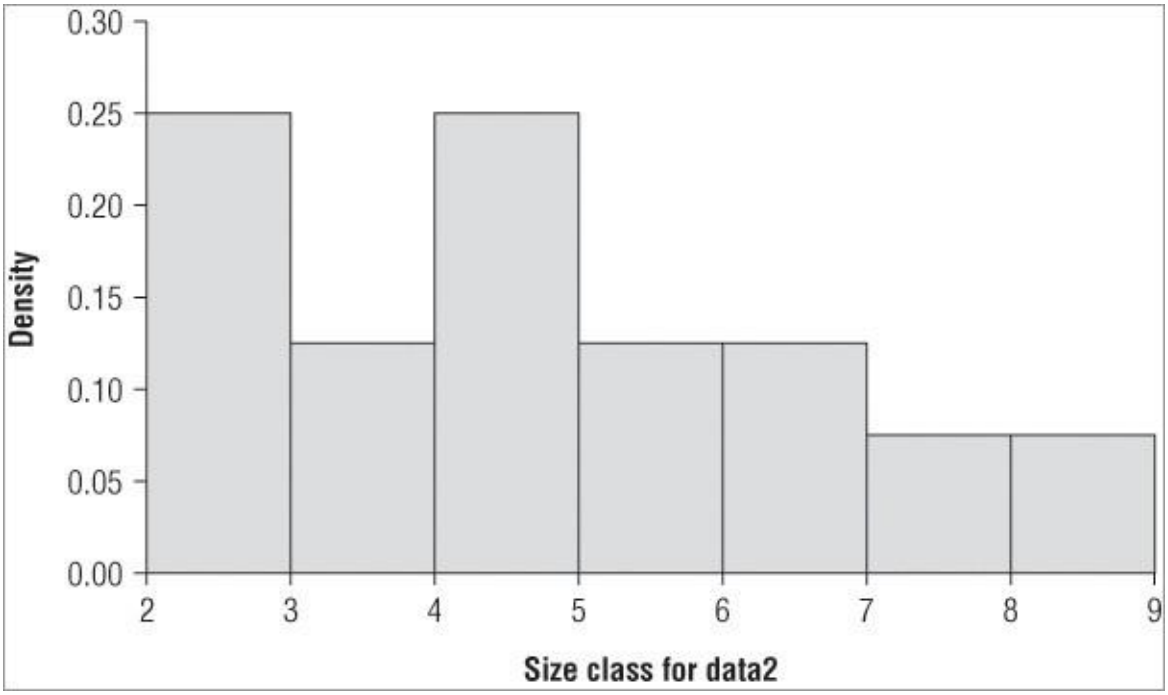
```
colours()
```

Note that this command does not have any additional instructions and you simply type the brackets. There are more than 650 colors to choose from!

As an example, here is a histogram created with a few additional instructions (see Figure5-3):

```
> hist(data2, col='gray75', main=NULL, xlab = 'Size class
for data2',
ylim=c(0, 0.3), freq = FALSE)
```

Figure 5-3



To create the histogram in Figure 5-3, perform the following steps:

1. Beginbyspecifyingthevectorofvaluesyourequire.

2. Next you use a new light gray color, `gray75`, before moving on to suppress the main title (you can add a title as a caption using your word processor).

3. Next, specify a title for the x-axis. The default scale of the y-axis in this case runs from 0 up to 0.25, but to give the axis a bit more room, alter the range from 0 to 0.3; you must always specify both the start and end points in theinstruction.

4. Lastly, change the plot from one of frequency to density by using `freq= FALSE`. You could specify these instructions in any order; the name of the vector of data usually goes first but you could put this later if you used `x = data.name`tospecifyitexplicitly.

You can even avoid a fair bit of typing if you omit the spaces and use a few abbreviations likeso:

```
> hist(co='gray75',ma=NULL,xla='Size class
for data2',yli=c(0,0.3),fr=F,x=data2)
```

However, if you show the command to anyone else they may not be able to recognize the

abbreviations. There is another (more sensible) reason to demonstrate the reordering of the command. If you want to create histograms for several vectors of data you can use the up arrow to recall the previous command. If the name of the data vector is at the end, it is quicker to edit it and to type the name of the new vector you want toplot.

NOTE

Every command has a set of possible instructions that it will accept. You can abbreviate the instructions you give as long as they are unique and unambiguous.

Density Function

You have seen in drawing a histogram with the `hist()` command that you can use `freq = FALSE` to force the y-axis to display the density rather than the frequency of the data. You can also call on the density function directly via the `density()` command. This enables you to draw your sample distribution as a line rather than a histogram. Many statisticians prefer the density plot to a regular histogram. You can also combine the two and draw a density line over the top of a regularhistogram.

You use the `density()` command on a vector of numbers to obtain the kernel densityestimateforthevectorinquestion.Theresultisaseriesof`x` and `y` coordinates that you can use to plot a graph. The basic form of the command is asfollows:

```
density(x, bw = 'nrd0', kernel = 'gaussian', na.rm = FALSE)
```

You specify your data, which must be a numerical vector, followed by the bandwidth. The bandwidth defaults to the `nrd0` algorithm, but you have several others to choose from or you can specify a value. The `kernel =` instruction enables you to select one of several smoothing options, the default being the "`gaussian`" smoother. You can see the various options from the help entry for this command. By default, `NA` items are not removed and an error will result if they are present; you can add `na.rm = TRUE` to ensure that you strip out any `NA` items.

If you use the command on a vector of numeric data you get a summary as a result like so:

```
> dens = density(data2)
> dens

Call:
        density.default(x = data2)

Data: data2(16obs.);       Bandwidth 'bw' =0.9644

       x                  y
 Min.   :-0.8932   Min.
        :0.0002982 1stQu.:2.3034
        1stQu.:0.0134042
 Median : 5.5000    Median :0.0694574
 Mean   : 5.5000   Mean    :0.0781187
 3rd Qu.: 8.6966 3rd Qu.:0.1396352
 Max.    :11.8932    Max. :0.1798531
```

The result actually comprises several items that are bundled together in a list object. You can see these items using the `names()` or `str()` commands:

```
> names(dens)
[1]"x"          "y"          "bw"         "n"
                "call" "data.name"
[7] "has.na"

>
str(dens)
List of 7
 $ x         : num [1:512] -0.893 -0.868 -0.843 -0.818 -0.793 ...
 $ y         : num [1:512] 0.000313 0.000339 0.000367 0.000397
0.000429 ...
 $ bw        : num 0.964
 $ n         : int 16
 $call      : language density.default(x =data2)
 $ data.name: chr "data2"
 $has.na    : logiFALSE
 - attr(*, "class")= chr "density"
```

You can extract the parts you want using `$` as you have seen with other lists. You might, for example, use the `$x`and `$y`parts to form the basis for a plot.

Using the Density Function to Draw a Graph

If you have a density result you can create a basic plot by extracting the `$x` and `$y` components and using them in a `plot()` command like so:

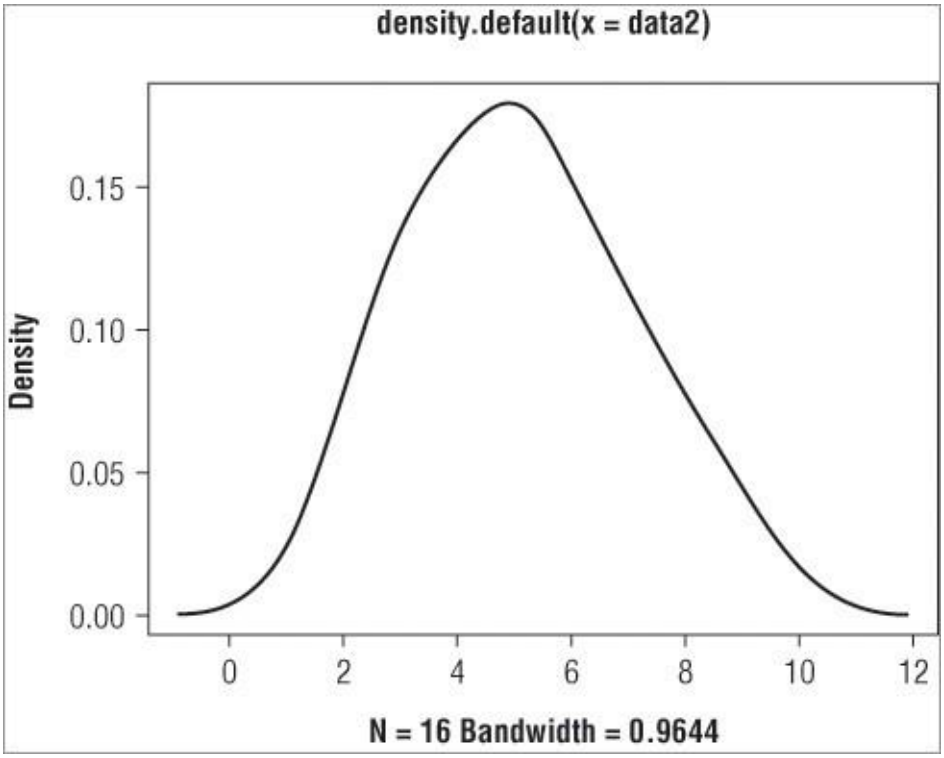
```
> plot(dens$x, dens$y)
```

However, for all practical purposes you do not need to go through any of this to produce a graph; you can use the `density()` command directly as part of a graphing command likeso:

```
> plot(density(data2))
```

This produces a graph like Figure 5-4.

Figure 5-4



The `plot()` command is a very general one in R and it can be used to produce a wide variety of graph types. In this case you see that the axes have been labeled and that there is also a main title. You can change these titles using the `xlab`, `ylab`, and `main` instructions as you saw previously. However, there is a slight difference if you want to remove a title completely. Previously you set `main = NULL` as your instruction, but this does not work here and you get the default title. You must use a pair of quotation marks so set the title to be empty:

```
> plot(density(data2), main = "", xlab = 'Size bin classes')
```

The preceding command removes the main title and alters the title of the x- axis. You can change other aspects of the graph as well; for instance, you already met the `xlim` and `ylim` instructions to resize the x and y axes, respectively.

Adding Density Lines to Existing Graphs

One use you might make for the density command is to add a density line to an existing histogram. Perhaps you want to compare the two methods of representation or you may want to compare two different samples; commonly one sample would be from an idealized distribution, like the normal distribution.

You can add lines to an existing graph using the `lines()` command. This takes a series of x and y coordinates and plots them on an existing graph. Recall earlier when you used the `density()` command to make an object called `dens`. The result was a list of several items including one called `x` and one called `y`. The

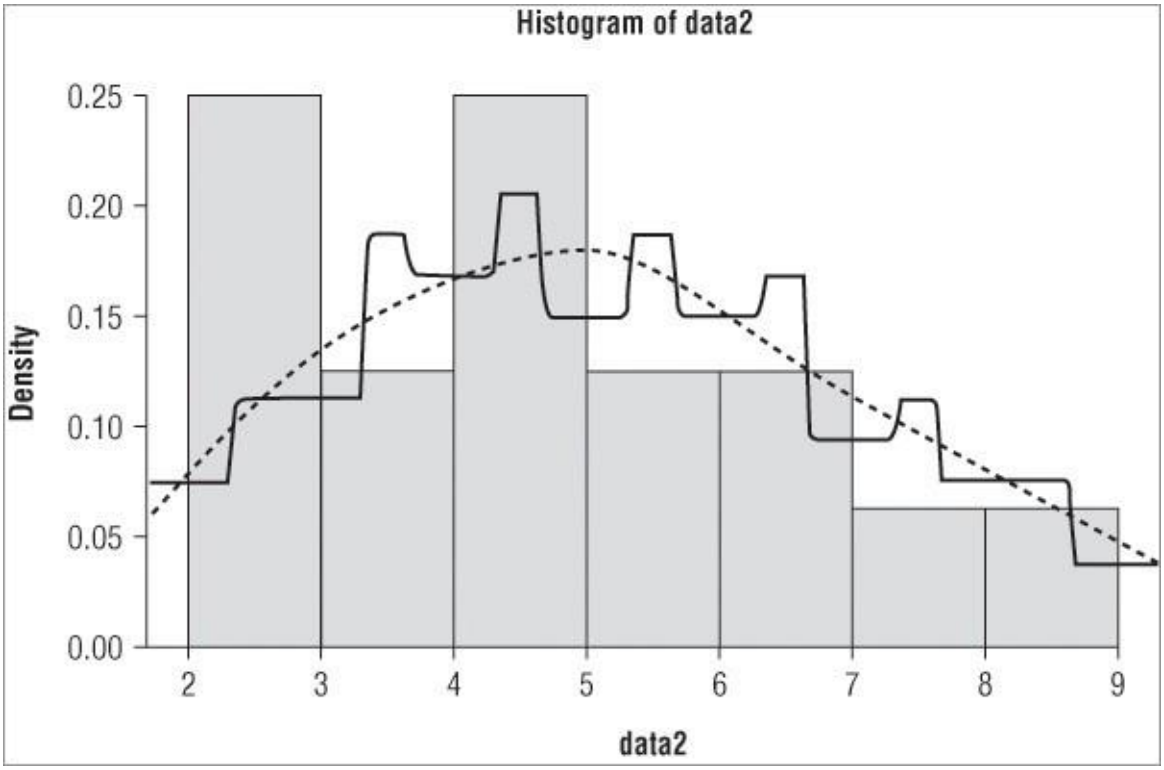
`lines()` commandcanreadthesetomaketheplot.

In the following example you produce a simple histogram and then draw two density lines over the top:

```
> hist(data2, freq = F, col = 'gray85')
> lines(density(data2), lty = 2)
> lines(density(data2, k = 'rectangular'))
```

In the first of the three preceding commands you produce the histogram; you must set the `freq = FALSE` to ensure the axis becomes density rather than frequency. The next two commands draw lines using two different density commands. The resulting graph looks like Figure 5-5.

Figure 5-5



Here you made the rectangular line using all the default options. The gaussian line has been drawn using a dashed line and this is done via the `lty =` instruction; you can use a numerical value where 1 is a solid line (the default), 2 is dashed, and 3 is dotted. Other options are shown in Table 5-2.

Table 5-2: Options for Line Style in the lty Instruction

Value	Label	Result
0	blank	Blank
1	solid	Solid (default)
2	dashed	Dashed
3	dotted	Dotted
4	dotdash	Dot-Dash
5	longdash	Long dash
6	twodash	Two dash

You can use either a numerical value or one of the text strings (in quotes) to produce the required effect; for example, `lty = "dotted"`. Notice that there is an option to have blank lines. It is also possible to alter the color of the lines drawn using the `col =` instruction. You can make the lines wider by specifying a magnification factor via the `lwd =`instruction.

Some additional useful commands include the `hist()` and `lines(density())`

commands with which you can draw an idealized distribution and see how your sample matches up. However, first you need to learn how to create idealized distributions.

Types of DataDistribution

You have access to a variety of distributions when using R. These distributions enable you to perform a multitude of tasks, ranging from creating random numbers based upon a particular distribution, to discovering the probability of a value lying within a certain distribution, or determining the density of a value for a given distribution. The following sections explain the many uses of distributions.

The Normal Distribution

Table 5-3shows the commands you can use in relation to the normal distribution.

Table 5-3: Commands Related to the Normal Distribution

Command	Explanation
`rnorm(n, mean = 0, sd = 1)`	Generates*n*randomnumbersfromthenormaldistributionwithmeanof0and standarddeviationof1
`pnorm(q, mean = 0, sd = 1)`	Returns the probability for the quantile `q`
`qnorm(p, mean = 0, sd = 1)`	Returns the quantile for a given probability `p`
`dnorm(x, mean = 0, sd = 1)`	Gives the density function for values `x`

You can generate random numbers based on the normal distribution using the `rnorm()` command; if you do not specify the mean or standard deviation the defaults of 0 and 1 are used. The following example generates 20 numbers with a mean of 5 and a standard deviation of 1:

```
> rnorm(20, mean = 5, sd = 1)
 [1] 5.610090 5.042731 5.120978 4.582450 5.015839 3.577376
5.159308 6.496983
 [9] 3.071729          5.02707  3.517274 4.393562
4.533490 6.021554
[17] 5.359491          3.81712  5.855315
```

You can work out probability using the `pnorm()` command. If you use the same mean and standard deviation as in the preceding code, for example, you might use the following:

```
> pnorm(5, mean = 5, sd =
1) [1] 0.5
```

In other words, you would expect a value of 5 to be halfway along the x-axis (your result is the cumulative proportion). By performing the following command you can turn this around and work out a value along the x-axis for any quantile;thatis,howfaralongtheaxisyouareasaproportionofitslength:

```
> qnorm(0.5, 5, 1)
```

```
[1] 5
```

You can see here that if you go 50 percent of the way along the x-axis you expect a value of 5, which is the mean of this distribution. You can also determine the density given a value. If you use the same parameters as the previous example did, you get thefollowing:

```
> dnorm(c(4,5,6), mean = 5, sd =
1) [1] 0.2419707 0.3989423
0.2419707
```

Here you work out the density for a mean of 5 and a standard deviation of 1.
This time you calculate the density for three values: 4, 5, and 6.

Additionally, you can use the `pnorm()` and `qnorm()` commands to determine one- and two-tailed probabilities and confidence intervals. For example:

```
> qnorm(c(0.05, 0.95), mean = 5, sd = 1)
[1] 3.355146 6.644854
```

Here is a situation in which it would be useful to compare the distribution of a sample of data against a particular distribution. You already looked at a sample of data and drew its distribution using the `hist()` command. You also used the `density()` command to draw the distribution in a different way, and to add the density lines over the original histogram. If you create a series of random numbers with the same mean and standard deviation as your sample, you can comparethe“ideal”normaldistributionwiththeactualobserveddistribution.

You can start by using `rnorm()` to create an ideal normal distribution using the mean and standard deviation of your data like so:

```
> data2.norm = rnorm(1000, mean(data2), sd(data2))
```

The more values in your distribution the smoother the final graph will appear, so here you create 1000 random numbers, taking the mean and standard deviation from the original data (called `data2`). You can display the two distributions in one of two ways; you might have the original data as the histogram and the idealized normal distribution as a line over the top, or you coulddrawtheidealnormaldistributionasahistogramandhaveyoursampleas
the line. The following shows the two options:

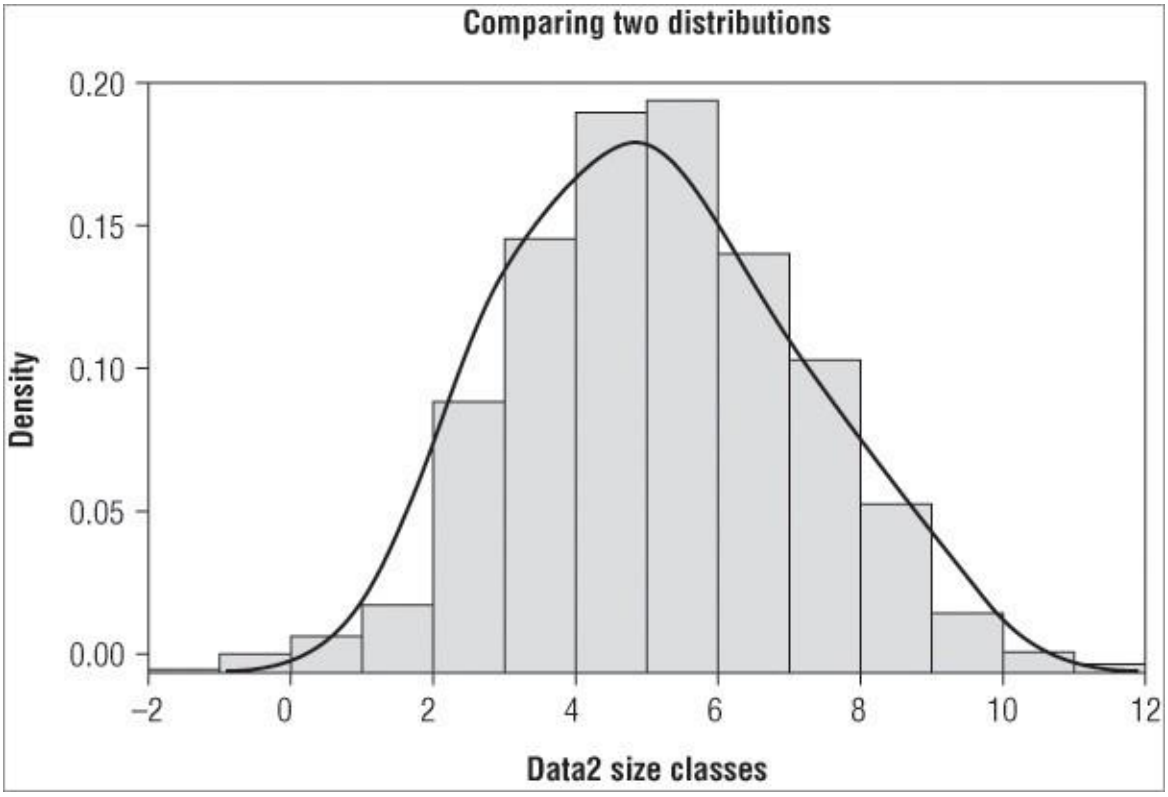
```
> hist(data2, freq = FALSE)
> lines(density(data2.norm))

> hist(data2.norm, freq = F)
> lines(density(data2))
```

In the first case you draw the histogram using your sample data and add the lines from the ideal normal distribution. In the second case you do it the other way around. With a bit of tweaking you can make a quite acceptable comparison plot. In the following example you make the ideal distribution a bit fainter by using the `border =` instruction. You also modify the x-axis and main titles. The linesrepresentingtheactualdataaredrawnandmadeabitbolderusingthe`lwd =`instruction. The resulting graph looks like Figure 5-6.

```
> hist(data2.norm, freq = F, border = 'gray50', main =
'Comparing two
distributions', xlab = 'Data2 size classes')
```

```
> lines(density(data2), lwd = 2)
```

Figure 5-6



You can see that you get a seemingly good fit to a normal distribution. Mathematical ways to compare the fit also exist, which you look at later in the chapter.

Other Distributions

You can use a variety of other distributions in a similar fashion; for full details look at the help entries by typing `help(Distributions)`. Look at a few examples now to get a flavor of the possibilities. In the following example, you start the Poisson distribution by generating 50 randomvalues:

```
> rpois(50, lambda = 10)
 [1] 10 12 10 13 10 11 81714    712   9 16  815   5   6  7 1011
151510    6 1012
[27]1411    7 12 1410   812   713   8  7  8   6  810  91212    5
11 12 11 12
```

The Poisson distribution has only a single parameter, `lambda`, equivalent to the mean. The next example uses the binomial distribution to assess probabilities:

```
> pbinom(c(3, 6, 9, 12), size = 17, prob = 0.5)
[1] 0.006363 0.166153 0.685471 0.975479
```

In this case you use `pbinom()` to calculate the cumulative probabilities in a binomial distribution. You have two additional parameters: `size` is the number of trials and `prob`is

the probability of each trial being a success.

You can use the Student's t-test to compare two normally distributedsamples. In this following example you use the `qt()` command to determine critical valuesforthe`t-test`forarangeofdegreesoffreedom.Youthengoontowork outtwo-sidedp-valuesforarangeoft-values:

```
> qt(0.975, df = c(5, 10, 100,
Inf)) [1] 2.571 2.228 1.984 1.960

> (1-pt(c(1.6, 1.9, 2.2), df =
Inf))*2 [1] 0.10960 0.05743 0.02781
```

In the first case you set the cumulative probability to 0.975; this will give you a 5 percent critical value, because effectively you want 2.5 percent of each end of the distribution (because this is a symmetrical distribution you can take 2.5 percent from each end to make your 5 percent). You put in several values for the *degrees of freedom* (related to the sample size). Notice that you can use `Inf` to represent infinity. The result shows you the value of `t` you would have to get for the differences in your samples (their means) to be significantly different at the 5 percent level; in other words, you have determined the criticalvalues.

In the second case you want to determine the two-sided p-value for various values of `t` when the degrees of freedom are infinity. The `pt()` commandwould

determine the cumulative probability if left to its own devices. So, you must subtract each one from 1 and then multiply by 2 (because you are taking a bit from each end of the distribution). You can get the same result using a modification of the command using the `lower.tail =` instruction. By default this is set to `TRUE`; this effectively means that you are reading the x-axis from left to right. If you set the instruction to `FALSE`, you switch around and read the x-axis from right to left. The upshot is that you do not need to subtract from one, which involves remembering where to place the brackets. The following example shows the results of the variousoptions:

```
> pt(c(1.6, 1.9, 2.2), Inf)
[1] 0.9452 0.9713 0.9861

> pt(c(1.6, 1.9, 2.2), Inf, lower.tail =
FALSE) [1] 0.05480 0.02872 0.01390

> pt(c(1.6, 1.9, 2.2), Inf, lower.tail =
FALSE)*2 [1] 0.10960 0.05743 0.02781
```

In the first result you do not modify the command at all and you get the standard cumulative probabilities. In the second case you use the `lower.tail= FALSE` instruction to get probabilities from "the other end." These probabilities areone-tailed(thatis,fromonlyoneendofthedistribution),soyoudoublethem togettherequired(two-tailed)values(asinthethirdcase).

In the next example you look at the F distribution using the `pf()` command. The F statistic requires two degrees of freedom values, one for the numerator and one for the denominator:

```
> pf(seq(3, 5, 0.5), df1 = 2, df2 = 12, lower.tail = F)
[1] 0.08779150 0.06346962 0.04665600 0.03481543 0.02633610
```

In this example you create a short sequence of values, starting from 3 and ending at 5 with an interval of 0.5; in other words, 3, 3.5, 4, 4.5, 5. You set the numerator `df` to 2 and the denominator to 12, and the result gives the cumulative probability. In this case it is perhaps easier to use the "other end" of the axis, so you set `lower.tail = FALSE` and get the p-values expressed as the "remainder" (thatis,thatpartofthedistributionthatliesoutsideyourcut-offpoint[s]).

The four basic commands `dxxx()`, `pxxx()`, `qxxx()`, and `rxxx()` provide you access to a range of distributions. They have common elements; the `lower.tail` = instruction is pretty universal, but each has its own particular instruction set. Table 5-4gives a sample of the distributions available; using `help(Distributions)` brings up the appropriate help entry.

Table 5-4: The Principal Distributions Available in R

Command	Distribution
`dbeta`	beta
`dbinom`	binomial (including Bernoulli)
`dcauchy`	Cauchy
`dchisq`	chi-squared
`dexp`	exponential
`df`	F distribution
`dgamma`	gamma
`dgeom`	geometric(specialcaseofnegativebinomial)
`dhyper`	hypergeometric
`dlnorm`	log-normal
`dmultinom`	multinomial
`dnbinom`	negative binomial
`dnorm`	normal
`dpois`	Poisson
`dt`	Student's t
`dunif`	uniform distribution
`dweibull`	Weibull
`dwilcox`	Wilcoxon rank sum
`ptukey`	Studentized range
`dsignrank`	Wilcoxon signed rank

The final distribution considered is the uniform distribution; essentially "ordinary" numbers. If you want to generate a series of random numbers, perhaps as part of a sampling exercise, use the `runif()` command:

```
> runif(10)
```

```
 [1] 0.65664996 0.58738275 0.07514039 0.34420863 0.30101891
0.58277238 0.24750941
 [8] 0.09282271 0.65748986 0.10004270
```

In this example `runif()` creates ten random numbers; by default the command uses minimum values of 0 and maximum values of 1, so the basic command produces random numbers between 0 and 1. You can set the `min` and `max` values with explicit instructions like so:

```
> runif(10, min = 0, max = 10)
 [1] 8.6480966 6.4076579 1.0365540 9.8101588 5.4944734
 8.2056503
4.2407627
 [8] 0.2206528 4.9709090 9.1819653
```

Now you have produced random values that range from 0 to 10. You can also use the density, probability, or quantile commands; use the following command to determine the cumulative probability of a value in a range of 0 to 10 (although you hardly needed the computer to work that oneout).

```
> punif(6, min = 0, max =
10) [1] 0.6
```

Random Number Generation and Control

R has the ability to use a variety of random number-generating algorithms (for more details, look at `help(RNG)` to bring up the appropriate help entry). You can alter the algorithm by using the `RNGkind()` command. You can use the command in two ways: you can see what the current settings are and you can also alter these settings. If you type the command without any instructions (that is, just a pair of parentheses) you see the currentsettings:

```
> RNGkind()
[1] "Mersenne-Twister" "Inversion"
```

Two items are listed. The first is the standard number generator and the second is the one used for normal distribution generation. To alter these, use the `kind =` and `normal.kind =` instructions along with a text string giving the algorithm you require; this can be abbreviated. The following example alters the algorithms and then resetsthem:

```
> RNGkind(kind = 'Super', normal.kind = 'Box')
> RNGkind()
[1] "Super-Duper" "Box-Muller"

> RNGkind('default')
> RNGkind()
[1] "Mersenne-Twister" "Box-Muller"

> RNGkind('default', 'default')
> RNGkind()
[1] "Mersenne-Twister" "Inversion"
```

Here you first alter both kinds of algorithm. Then you query the type set by running the

command without instructions. If you use `default` as an instruction, you reset the algorithms to their default condition. However, notice that you have to do this for each kind, so to restore the random generator fully you need two `default` instructions, one for eachkind.

There may be occasions when you are using random numbers but want to get the same random numbers every time you run a particular command.Common

examples include when you are demonstrating something or testing and want to get the same thing time and time again. You can use the `set.seed()` command to dothis:

```
> set.seed(1)
> runif(1)
[1]0.2655087

> runif(1)
[1]0.3721239

> runif(1)
[1]0.5728534

> set.seed(1)
> runif(3)
[1] 0.2655087 0.3721239 0.5728534
```

You use a single integer as the instruction, which sets the starting point for random number generation. The upshot is that you get the same result every time. In the preceding example you set the seed to 1 and then use three separate commands to create three random numbers. If you reset the seed to 1 and generate three more random numbers (using only a single command this time) you get the same values!

You can also use the `set.seed()` command to alter the kind of algorithm using the `kind =` and `normal.kind =` instructions in the same way you did when using the `RNGkind()` command:

```
> set.seed(1, kind = 'Super')
> runif(3)
[1] 0.3714075 0.4789723 0.9636913

> RNGkind()
[1] "Super-Duper" "Inversion"

> set.seed(1, kind = 'default')
> runif(3)
[1] 0.2655087 0.3721239 0.5728534

> RNGkind()
[1] "Mersenne-Twister" "Inversion"
```

In this example you set the seed using a value of 1 (you can also use negative values) and altered the algorithm to the Super-Duper version. After you use this to make three random numbers you look to see what you have before setting the seedto1againbutalsoresettingthealgorithmtoitsdefault,theMersenne-Twister.

Random Numbers and Sampling

Another example where you may require randomness is in sampling. For instance, if you have

a series of items you may want to produce a random smaller sample from these items. See the following example in which you have a simple vector of numbers and want to choose four of these to use for some other purpose:

```
> data2
 [1] 3 5 7 5 3 2 6 8 5 6 9 4 5 7 3 4
> sample(data2, size =
4) [1] 3 8 9 6
```

In this example you extract four of your values as a separate sample from the `data2` vector of values. You can do this for character vectors as well; in the following example you have a character vector that comprises 12 months. You use the `replace =`instruction to decide if you want replacement or not like so:

```
> sample(data8, size = 4, replace =
TRUE) [1] "Apr" "Jan" "Feb" "Oct"

> sample(data8, size = 4, replace =
TRUE) [1] "Feb" "Feb" "Jun" "May"
```

Here you set `replace = TRUE` and the effect is to allow an item to be selected more than once. In the first example all four items are different, but in the second case you get the `Feb` result twice. The default is to set `replace = FALSE`, resulting in items not being selected more than once; think of this as not replacing an item in the result vector once ithas been selected (placed).

You can extend this and select certain conditions to be met from your sampled data. In the following example you pick out three items from your original data but ensure that they are all greater than5:

```
> data2
 [1] 3 5 7 5 3 2 6 8 5 6 9 4 5 7 3 4
> sample(data2[data2 > 5], size =
3) [1] 7 9 8
```

Ifyouleavethe`size=`partout,yougetasampleofeverythingthatmeetsany conditionsyouhaveset:

```
> sample(data2[data2 >
5]) [1] 9 8 7 6 6 7

> data2[data2 > 5]
[1] 7 6 8 6 9 7
```

In the first case you randomly select items that are greater than 5. In the second case you display all the items that are greater than 5. When you merely display the items they appear in the order they are in the vector, but when you use the `sample()` command they appear in random order. You can see this clearly by using the same command severaltimes:

```
>                    > 5])
[1] 8 6 7 9 7 6
>                    > 5])
[1] 6 9 7 8 7 6
```

```
>                     > 5])
[1] 7 7 9 6 8 6
```

Because of the way the command is programmed you can get an unusual result:

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

You might have expected to get a single result (of 9) but you do not. Instead, your condition has resulted in 1 (there is only 1 item greater than 8). You are essentially picking out items that range from 1 to > 8. Because there is onlyone item>8yougetasamplefrom1to9,andifyoulookyouseenineitemsinyour result.Inthefollowingexample,youlookforitems>9:

```
> data3
 [1]  6  7  8  7  6  3  8  910   7  6  9
> sample(data3[data3 > 9])
 [1]  1  5  4  7  610   3  8  9  2
```

Because there is only one of them (a 10) you get ten items in your result sample. This is slightly unfortunate but there is a way around it, which is demonstrated in the help entry for the `sample()` command. You can create a simple function to alter the way the `sample()` command operates; first type the following like so:

```
> resample <- function(x, ...) x[sample(length(x), ...)]
```

This creates a new command called `resample()`, which you use exactly like youwouldtheold`sample()` command.Yournewcommand,however,givesthe "correct"result;thefollowingexampleshowsthecomparisonbetweenthetwo:

```
> data2
 [1] 3 5 7 5 3 2 6 8 5 6 9 4 5 7 3 4
> set.seed(4)
> sample(data2, size =
3) [1] 2 6 7

> set.seed(4)
> resample(data2, size =
3) [1] 2 6 7

> set.seed(4)
> sample(data2[data2 >
8]) [1] 3 5 2 8 4 7 1 9 6

> set.seed(4)
> resample(data2[data2 >
8]) [1] 9
```

In this example you use the `set.seed()` command to ensure that your random numbers come out the same; you use a value of 4 but this is merely a whim, any integer will suffice. At

the top you can see that you get exactly the same result when you extract three random items as a sample from your original vector. When you use the `sample()` command to extract values > 8 you see the error. However, you see at the bottom that the `resample()` command has given the expected result.

Creating simple functions is quite straightforward; in the following case the function is named `resample` and this appears as an object when you type an `ls()` command:

```
> ls(pattern = '^resa')
[1] "resample"
"response"
> str(resample)
function (x, ..)
 - attr(*, "source")= chr "function(x, ...) x[sample(length(x),
...)]"
```

In this example you choose to list objects beginning with "res" and see two objects. If you use the `str()` command you can see that the `resample` object is a function. If you type the name of the object you get to see more clearly what it does—you get the code used to create it:

```
> resample
function(x, ...) x[sample(length(x), ...)]

>
class(resample)
[1] "function"
```

The left part shows the instructions expected; here you have `x` and three dots.

The `x` simply means a name, the vector you want to sample, and the three dots mean that you can use extra instructions that are appropriate. The right part shows the workings of the function; you see your object represented as `x`, and the `sample()` and `length()` commands perform the actual work of the command. The final three dots get replaced by whatever you type in the command as an extra instruction; you used the `size =` instruction in one of the previous examples. In the following examples you use another appropriate instruction and one inappropriateone:

```
> resample(data2[data2 > 8], size = 2, replace =
T) [1] 9 9

> resample(data2[data2 > 8], size = 2, replace = T, na.rm =
T) Error in sample(length(x), ...) : unused argument(s)
(na.rm = TRUE)
```

In the first case the `replace = TRUE` instruction is valid because it is used by `sample()`. In the second case, however, you get an error because the `na.rm =` instruction is not used by either `sample()` or `length()`.

Creating functions is a useful way to unlock the power of R and enables you to create templates to carry out tedious or involved commands over and over again with minimal effort. You look at the creation of custom functions in more detail in Chapter10.

The Shapiro-Wilk Test for Normality

You commonly need to compare a sample with the normal distribution. You saw previously how you could do this graphically using a histogram and a density plot. There are other graphical methods, which you will return to shortly, but there are also statistical methods. One such method is the Shapiro-Wilk test, which is available via the `shapiro.test()` command. Using it is very simple; just provide the command with a numerical vector to work on:

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

> shapiro.test(data2)

        Shapiro-Wilk normality test

data:  data2
W = 0.9633, p-value = 0.7223
```

The result shows that the sample you have is not significantly different from a normal distribution. If you create a sample using random numbers from another (not normal) distribution, you would expect a significant departure. In the following example you use the `rpois()` command to create 100 random values from a Poisson distribution with `lambda`set to 5:

```
> shapiro.test(rpois(100, lambda = 5))

        Shapiro-Wilk normality test

data:  rpois(100, lambda =5)
W = 0.9437, p-value = 0.0003256
```

You see that you do get a significant departure from normality in this case. If you have your data contained within another item, you must extract it in some way. In the following example you have a data frame containing three columns, two of numeric data and one ofcharacters:

```
> grass3
  rich graze poa
1   12   mow   4
2   15   mow   5
3   17   mow   6
4   11   mow   5
5   15   mow   4
6    8 unmow   5
7    9 unmow   6
8    7 unmow   8
9    9 unmow   7
> shapiro.test(grass3$rich)

        Shapiro-Wilk normality test
```

```
data:  grass3$rich
W = 0.9255, p-value = 0.4396
```

In this case you use the `$` to get the `rich` sample as your vector to compare to normality. You probably ought to test each grazing treatment separately, so you have to subset a little further like so:

```
> with(grass3, shapiro.test(rich[graze == 'mow']))

        Shapiro-Wilk normality test

data:  rich[graze =="mow"]
W = 0.9251, p-value = 0.5633
```

In this example you use the `with()` command has saved you a little bit of typing by enabling you to read inside the `grass` data frame temporarily. When you have only a couple of treatment levels this is not too tedious, but when you have several it can become a chore to repeat the test for every level, even if you can use the up arrow to recall the previous command. There is a way around this; you will see the command in more detail in Chapter 9, but for now an example will suffice:

```
> tapply(grass3$rich, grass3$graze, shapiro.test)
$mow

        Shapiro-Wilk normality test

data:  X[[1L]]
W = 0.9251, p-value = 0.5633
$unmow

        Shapiro-Wilk normality test

data:  X[[2L]]
W = 0.8634, p-value = 0.2725
```

In this example you use the command `tapply()`, which enables you to cross- tabulate a data frame and apply a function to each result. The command starts with the column you want to apply the function to. Then you provide an index to carry out the cross tabulation; here you use the `graze` column. You get two results because there were two levels in the `graze` column. At the end you specify the command/function you want to use and you caN also add extra instructions if they are applicable. The result is that the `shapiro.test()` is applied to each combination of `graze` for your `rich` data.

The Kolmogorov-Smirnov Test

The Kolmogorov-Smirnov test enables you to compare two distributions. This means that you can either compare a sample to a "known" distribution or you can compare two unknown distributions to see if they are the same; effectively you are comparing the shape

```
> ks.test(data2, 'pnorm', mean = 5, sd = 2)
```

```
        One-sample Kolmogorov-Smirnov

data:  data2
D = 0.125, p-value = 0.964
alternative hypothesis: two-
sided

Warning message:
In ks.test(data2, "pnorm", mean = 5, sd = 2)
  : cannot compute correct p-values with
  ties
```

In this case you specify the cumulative distribution function you want as a text string (that is, in quotes) and also give the required parameters for the normal distribution; in this case the mean and standard deviation. This carries out a one- sample test because you are comparing to a standard distribution. Note, too, that you get an error message because you have tied values in your sample. You could create a normal distributed sample "on the fly" and compare this to your sample like so:

```
> ks.test(data2, pnorm(20, 5,
2))

        Two-sample Kolmogorov-Smirnov

data:  data2 and pnorm(20, 5,2)
D = 1, p-value = 0.3034
alternative hypothesis: two-
sided

Warning message:
In ks.test(data2, pnorm(20, 5, 2)) :
  cannot compute correct p-values
  withties
```

Now in this example you have run a two-sample test because you have effectively created a new sample using the `pnorm()` command. In this case the parameters of the normal distribution are contained in the `pnorm()` command itself. You can also test to see if the distribution is less than or greater than your comparison distribution by adding the `alternative =` instruction; you use `less` or `greater`because the default is `two.sided`.

You can, of course, use other distributions; the following example compares a data sample to a Poisson distribution (with `lambda = 5`):

```
> ks.test(data2, 'ppois', 5)
        One-sample Kolmogorov-Smirnov test

data:  data2
D = 0.241, p-value = 0.3108
```

```
alternative hypothesis: two-
sided

Warning message:
In ks.test(data2, "ppois", 5) : cannot compute correct p-
values with ties
```

Quantile-Quantile Plots

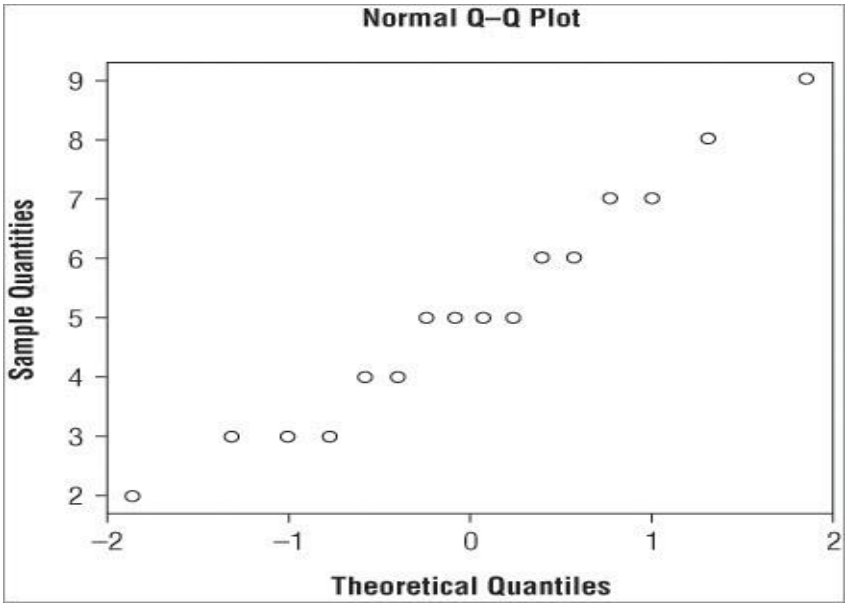
Earlier you looked at histograms and density plots to visualize a distribution; you can perhaps estimate the appearance of a normal distribution by its bell-shaped appearance. However, it is easier to judge if you can get your distribution to lie in a straight line. To do that you can use quantile-quantile plots (QQ plots). Many statisticians prefer QQ plots over strictly mathematical methods like the Shapiro-Wilk test for example.

A Basic Normal Quantile-Quantile Plot

You have several commands available relating to QQ plots; the first of these is `qqnorm()`, which takes a vector of numeric values and plots them against a set of theoretical quantiles from a normal distribution. The upshot is that you produce a series of points that appear in a perfectly straight line if your original data are normally distributed. Run the following command to create a simple QQ plot (the graph is shown in Figure5-7):

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

Figure 5-7



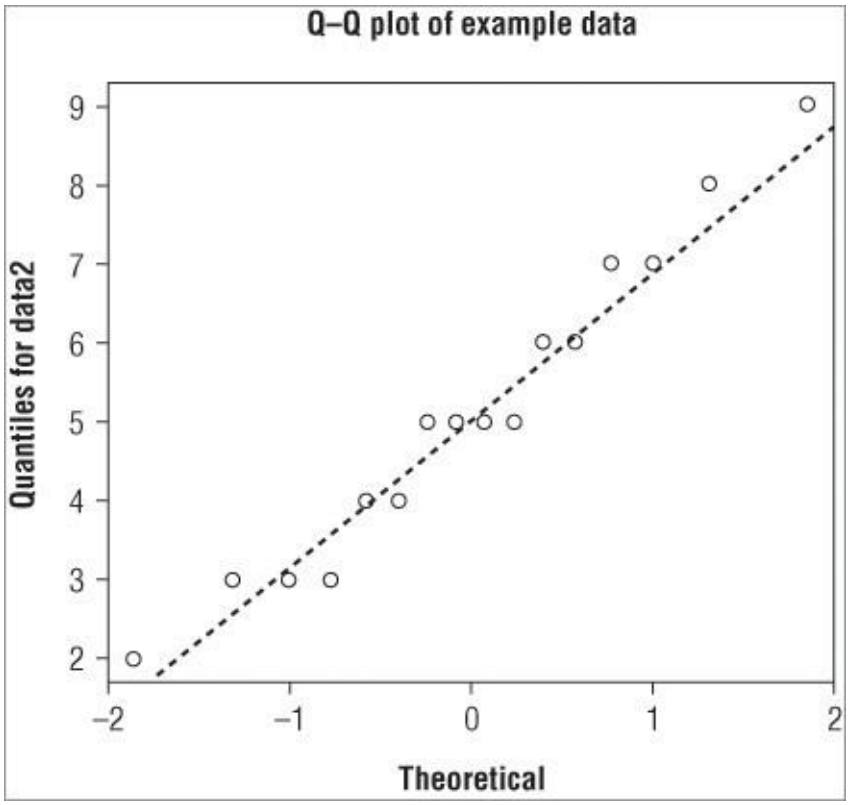
The main title and the axis labels have default settings, which you can alter by using the `main`, `xlab`, and `ylab`instructions that you met previously.

Adding a Straight Line to a QQ Plot

The normal QQ plot is useful for visualizing the distribution because it is easier to check alignment along a straight line than to check for a bell-shaped curve. However, you do not currently have a straight line to check! You can add one using the `qqline()` command. This adds a straight line to an existing graph. You can alter the appearance of the line using various instructions; you can change the color, width, and style using `col`, `lwd`, and `lty` instructions (which you have met previously). If you combine the `qqnorm()` and `qqlines()` commands you can make a customized plot; the following example produces a plot that looks like Figure5-8:

```
> qqnorm(data2, main = 'QQ plot of example data', xlab =
'Theoretical',
ylab = 'Quantiles for data2')
> qqline(data2, lwd = 2, lty = 2)
```

Figure 5-8



Plotting the Distribution of One Sample Against Another

You can also plot one distribution against another as a quantile-quantile plot using the `qqplot()` command. To use it you simply provide the command with the names of the two distributions you want tocompare:

```
> qqplot(rpois(50,5), rnorm(50,5,1))
> qqplot(data2, data1)
```

In the top example you compare two distributions that you create “on the fly” from random

numbers. The bottom example compares two samples of numeric data; note that this is *not* simply one sample plotted against the other (because they have different lengths you could not do that anyhow). It would be useful to draw a straight line on your `qqplot()` and you can do that using the `abline()` command. This command uses the properties of a straight line (that is, `y = a + bx`)toproducealineonanexistingplot.Thegeneralformofthecommandis:

```
abline(a = intercept, b = slope)
```

You supply the intercept and slope to produce the straight line. The problem here is that you do not know what the intercept or slope values should be! You need to determine these first; fortunately the `abline()` command can also use results from other calculations. In the following example you take the `data2` sample and compare this to a randomly-generated normal distribution with 50 values; you set the mean to 5 and the standard deviation to2:

```
> qqp = qqplot(data2, rnorm(50,5,2))
```

This makes a basic plot; your sample is on the x-axis and the sample you compare to, the random one, is on the y-axis. Notice that you do not just make the plot but assign it to an object; here called `qqp`. You do this because you want to see the values used to create the plot. If you type the name of your plot (`qqp`) you see that you have a series of x-values (your original `data2`) and a series of y- values:

```
> qqp
$x
 [1] 2 3 3 3 4 4 5 5 5 5 6 6 7 7 8 9

$y
 [1] 1.405236 2.625890 3.429247 4.037570 4.433178 4.648895
4.983500 5.292363
 [9] 5.372463 6.154243 6.424723 6.817186 7.360115 7.580486
7.976507 8.793080
```

The `qqplot()` command has used the distribution you created (using the `rnorm()` command) as the basis to generate quantiles for the y-axis. Now the `x` and `y` values match up. You can use these values to determine the intercept and slope and then draw your straight line:

```
> abline(lm(qqp$y ~ qqp$x))
```

Another new command must be introduced at this point: `lm()`, which carries outlinearmodeling.Thiscommanddeterminesthelineofbestfitbetweenthe`x` and `y` values in your `qqp` object. The `abline()` command is able to read the result directly so you do not have to specify the `a =` and `b =` instructions explicitly.

So, the final set of commands appears like this:

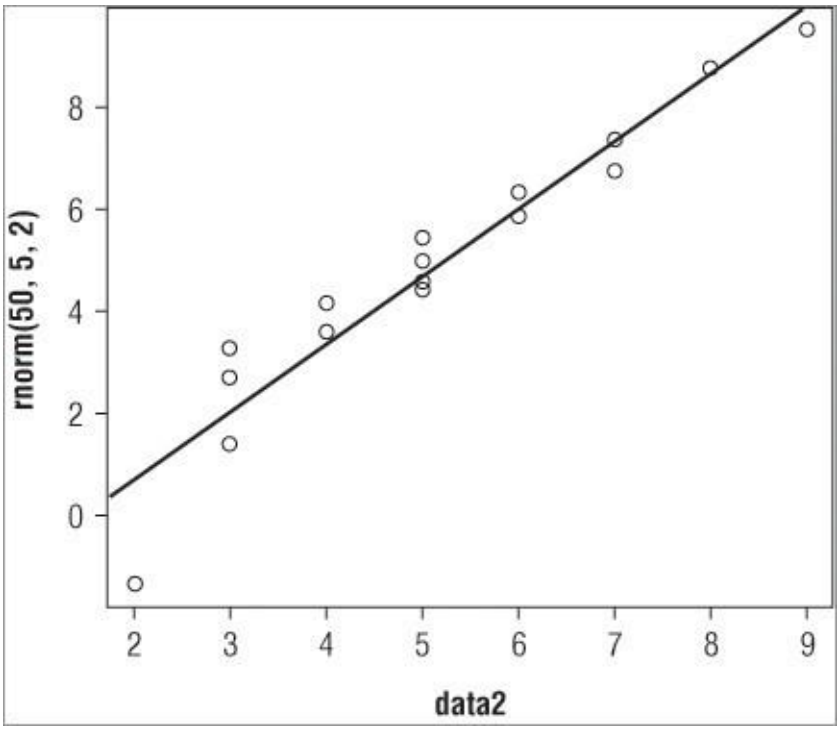
```
> qqp = qqplot(data2, rnorm(50, 5, 2))
> abline(lm(qqp$y ~ qqp$x))
```

This produces the plot shown in Figure 5-9.

You can alter the titles of the plot and the appearance of the line using the same commands that you met previously (`main`, `xlab`, `ylab`, `lwd`, `lty`, and `col`).

You look at graphical commands in more detail in Chapter 7 and also Chapter **11.** You also look more carefully at the `lm()` command (in Chapter 10), which is used in a wide range of statisticalanalyses.

Figure 5-9



Summary

- You can visualize the distribution of a numeric sample using the `stem()` command to make a stem-leaf plot or the `hist()` command to draw a histogram.
- A variety of distributions can be examined with R; these include the normal, Poisson, and binomial distributions. The distributions can be examined with a variety of commands (for example, `rfoo()`, `pfoo()`, `qfoo()`, and `dfoo()`, where `foo`is the distribution).
- You can test the distribution of a sample using the Shapiro-Wilk test for normality via the `Shapiro.test()` command. The Kolmogorov-Smirnov test can compare two distributions and is accessed via the `ks.test()` command.
- Quantile-Quantile plots can be produced using `qqnorm()` and `qqplot()` commands. The `qqlines()` command can add a straight line to a normal QQ plot.

Exercises

You can find the answers to these exercises in Appendix A.

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

1. Examine the `orchis2` data object. Here you see a two-column data frame with a

response variable (`flower`) and a predictor variable (`site`). Produce a histogram for the `sprayed`site. Now overlay a density plot.

2. Determine the critical value (at 5% and 1% two-tailed) of the Wilcoxon statistic for a two-sample test where n = 8 for both samples. If you carried out a Wilcoxon two-sample test, where each sample contained ten replicates, and got a result of 77, how could you determinethe statistical significance?

What You Learned in This Chapter

Topic	Key Points
Numerical distribution (for example, norm, pois, binom, chisq, Wilcox, unif, t, F): **`rfoo()` `pfoo()` `qfoo()` `dfoo()`**	Manydistributionsareavailableforanalysisin R. Theseincludethenormaldistribution as well as Poisson, binomial, and gamma. Other statistical distributions include chi-squared, Wilcox, F, and t.Four main commands handle distributions. The `rfoo()` commandgeneratesrandomnumbers(where `foo`isthedistribution),`pfoo()` determines probability, `dfoo()` determines density function, and `qfoo()` calculates quantiles.
Random numbers: **`RNGkind()` `set.seed()` `sample()`**	Random numbers can be generated for many distributions. The `runif()` command, for example,createsrandomnumbersfromtheuniformdistribution.Avarietyofalgorithms to generate random numbers can be used and set via the `RNGkind()` command. The `set.seed()` commanddeterminesthe"startpoint"forrandomnumbergeneration.The `sample()` commandselectsrandomelementsfromalargerdatasample.
Drawing distribution: `stem()` `hist()` `density()` `lines()` `qqnorm()` `qqline()` `qqplot()`	The distribution of a numerical sample can be drawn and visualized using several commands:the`stem()` commandcreatesasimplestemandleafplot,forexample.The `hist()` command draws classic histograms, and the `density()` command allows the densitytobedrawnontoagraphviathe`lines()` command.Quantile-quantileplotscan bedealtwithusing`qqnorm()`,whichplotsadistributionagainstatheoreticalnormal.A linecanbeaddedusing`qqline()`.The`qqplot()` commandenablestwodistributionsto beplottedagainstoneanother.
Testing distribution: **`shapiro.test()` `ks.test()`**	The normality of a distribution can be tested using the Shapiro-Wilk test via the `shapiro.test()` command. The Kolmogorov-Smirnov test can be used via the `ks.test()` command.Thiscantestonedistributionagainstaknown"standard"orcan testtoseeiftwodistributionsarethesame.
Graphics: **`plot()` `abline()` `lines()` `lty lwdcol` `xlim xlab ylim ylab main` `colors()` `lm()`**	The `plot()` command is a very general graphical command and is often called in the background as the result of some other command (for example, `qqplot()`).The `abline()` commandaddsstraightlinestoexistinggraphsandcanusetheresultof previous commands to determine the coordinates to use (for example, the `lm()` command,whichdeterminesslopeandinterceptintherelationshipbetweentwo variables).The `lines()` command adds sections of line to an existing graph and can be usedtoaddadensityplottoanexistinghistogram.Manyadditionalparameterscanbe addedto`plot()` andothergraphicalcommandstoprovidecustomization.Forexample, `col`alters the color of plotted elements, and `xlim`alters the limits of the x-axis. A comprehensive list can be found by using `help(par)`.The `colors()` command gives a simple list of the colors available.

Chapter 6

Simple Hypothesis Testing

What you will learn in this chapter:

- How to carry out some basic hypothesis tests How to
- carry out the Student's t-test
- How to conduct the U-test for non-parametric data
- How to carry out paired tests for parametric and non-parametric data How to
- produce correlation and covariance matrices
- How to carry out a range of correlations tests How to
- test for association using chi-squared How to carry
- out goodness of fit tests

Many statistical analyses are concerned with testing hypotheses. In this chapter you look at methods of testing some simple hypotheses using standard and classic tests. You start by comparing differences between two samples. Then you look at the correlation between two samples, and finally look at tests for association and goodness of fit. Other tests are available in R, but the ones illustrated here will form a good foundation and give you an idea of how R works. Should you require a different test, you will be able to work out how to carry it out foryourself.

Using the Student'st-test

The Student's t-test is a method for comparing two samples; looking at the means to determine if the samples are different. This is a parametric test and the data should be normally distributed. You looked at the distribution of data previously in Chapter5.

Several versions of the t-test exist, and R can handle these using the `t.test()` command, which has a variety of options (see Table 6-1), and the test can be pressed into service to deal with two- and one-sample tests as well as paired tests. The latter option is discussed in the later section "Paired T- and U-Tests";

in this section you look at some more basic options.

Table 6-1: The t.test() Command and Some of the Options Available.

Command	Explanation
`t.test(data.1, data.2)`	The basic method of applying a t-test is to compare two vectors of numeric data.
`var.equal = FALSE`	If the `var.equal` instruction is set to `TRUE`, the variance is considered to be equal and the standard test is carried out. If the instruction is set to `FALSE` (the default), the variance is considered unequal and the Welch two-sample test is carried out.
`mu = 0`	If a one-sample test is carried out, `mu` indicates the mean against which the sample should be tested.
`alternative = "two.sided"`	Sets the alternative hypothesis. The default is `"two.sided"` but you can specify `"greater"` or `"less"`. You can abbreviate the instruction (but you still need quotes).

`conf.level = 0.95`	Sets the confidence level of the interval (default = 0.95).
`paired = FALSE`	If set to `TRUE`, a matched pair t-test is carried out.
`t.test(y ~ x, data, subset)`	Therequireddatacanbespecifiedasaformulaoftheform`response~predictor`. In this case, the data should be named and a subset of the predictor variable can be specified.
`subset = predictor %in% c("sample.1", "sample.2")`	Ifthedataisintheform`response~predictor`,thesubsetinstructioncanspecify whichtwosamplestoselectfromthepredictorcolumnofthedata.

Two-Sample t-Test with UnequalVariance

The general way to use the `t.test()` command is to compare two vectors of numeric values. You can specify the vectors in a variety of ways, depending how your data objects are set out. The default form of the `t.test()` does not assume that the samples have equal variance, so the Welch two-sample test is carried out unless you specify otherwise:

```
> t.test(data2, data3)

      Welch Two Sample t-test

data:  data2 anddata3
t = -2.8151, df = 24.564, p-value = 0.009462
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 -3.5366789 -0.5466544
sample estimates:
mean of x mean of y
 5.125000  7.166667
```

Two-Sample t-Test with EqualVariance

You can override the default and use the classic t-test by adding the `var.equal = TRUE` instruction, which forces the command to assume that the variance of the two samples is equal. The calculation of the t-value uses pooled variance and the degrees of freedom are unmodified; as a result, the p-value is slightly different from the Welch version:

```
> t.test(data2, data3, var.equal = TRUE)

      Two Sample t-test

data:  data2 anddata3
t = -2.7908, df = 26, p-value = 0.009718
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 -3.5454233 -0.5379101
sample estimates:
mean of x mean of y
 5.125000  7.166667
```

One-Sample t-Testing

You can also carry out a one-sample t-test. In this version you supply the name of a single vector and the mean to compare it to (this defaults to0):

```
> t.test(data2, mu = 5)

      One Sample t-test

data:  data2
t = 0.2548, df = 15, p-value = 0.8023
alternative hypothesis: true mean is not equal to 5
95 percent confidence interval:
 4.079448 6.170552
sample estimates:
mean of x
    5.125
```

Using Directional Hypotheses

You can also specify a "direction" to your hypothesis. In many cases you are simply testing to see if the means of two samples are different, but you may want toknowifasamplemeanislowerthananothersamplemean(orgreater).You can use the `alternative =` instruction to switch the emphasis from a two-sided test (the default) to a one-sided test. The choices you have are between "`two.sided`", "`less`", or "`greater`", and your choice can be abbreviated.

```
> t.test(data2, mu = 5, alternative = 'greater')

      One Sample t-test

data:  data2
t = 0.2548, df = 15, p-value = 0.4012
alternative hypothesis: true mean is greater than 5
95 percent confidence interval:
 4.265067      Inf
sample estimates:
mean of x
    5.125
```

Formula Syntax and Subsetting Samples in the t-Test

The t-test is designed to compare two samples (or one sample with a "standard"). So far you have seen how to carry out the t-test on separate vectors of values. However, your data may well be in a more structured form with a column for the response variable and a column for the predictor variable. The following data are set out in this manner:

```
> 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
```

This way of setting out data is more sensible and flexible, but you need a new way to deal with the layout. R deals with this by having a "formula syntax." You create a formula using the tilde (~) symbol. Essentially your response variable goes on the left of the ~and the predictor goes on the right like so:

```
> t.test(rich ~ graze, data = grass)
      Welch Two Sample t-test

data:  rich bygraze
t = 4.8098, df = 5.411, p-value = 0.003927
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 2.745758 8.754242
sample estimates:
  mean in group mow mean in group unmow
              14.00                8.25
```

If your predictor column contains more than two items, the t-test cannot be used. However, you can still carry out a test by subsetting this predictor column and specifying which two samples you want to compare. You must use the `subset =` instruction as part of the `t.test()` command. The following example illustrateshowtodothisusingthesamedataasinthepreviousexample.

```
> t.test(rich ~ graze, data = grass, subset = graze %in% c('mow',
'unmow'))
```

You first specify which column you want to take your subset from (`graze` in this case) and then type `%in%`; this tells the command that the list that follows is contained in the `graze` column. Note that you have to put the levels in quotes; here you compare "`mow`" and "`unmow`" and your result (not shown) is identical to that you obtainedbefore.

Try It Out: Carry Out Student's t-Tests on SomeData

Use the data on orchids (orchid, orchid2, orchis, and orchis2) from theBeginning.RData file for this activity, on which you will be carrying out a range oft-tests.

1. Use the ls() command to see the data you require; they all begin with"orchi":
   > ls(pattern='^orc')
   [1]"orchid" "orchid2""orchis" "orchis2"

2. Look first at the orchid data. This comprises two columns relating to twosamples:

```
> orchid closed open
1          7      3
2          8      5
 3         6      6
 4         9      7
 5        10      6
 6        11      8
 7         7      8
 8         8      4
 9        10      7
10         9      6
```

3. Carry out a t-test on these data without making any assumptions about the variance, like so:

```
> attach(orchid)
> t.test(open, closed)

    Welch Two Sample t-test

data:  open andclosed
t = -3.478, df = 17.981, p-value = 0.002688
alternative hypothesis: true difference in means is not equal to0
95 percent confidence interval:
   -4.0102455 -0.9897545
sample estimates: mean of
x mean of y
        6.0         8.5

> detach(orchid)
```

4. Now carry out another two-sample t-test but use the "classic" version and assume the variance of the two samples isequal:

```
> with(orchid, t.test(open, closed, var.equal = TRUE))
     Two Sample t-test

data:  open andclosed
t = -3.478, df = 18, p-value = 0.002684
alternative hypothesis: true difference in means is not equal to0
95 percent confidence interval:
   -4.0101329 -0.9898671
sample estimates: mean of
x mean of y
        6.0         8.5
```

5. This time look at the open sample only and carry out a one-sample test to compare the data to a mean of5:

```
> t.test(orchid$open, mu = 5)

     One Sample t-test data:

          orchid$open
t = 1.9365, df = 9, p-value = 0.08479
alternative hypothesis: true mean is not equal to 5
95 percent confidence interval:
 4.831827 7.168173
```

sample estimates: mean
of x
6
Now look at the orchis data object. It has two columns, flower and site. Use the str() or summary() command to confirm that there aretwo samples in the sitecolumn:

```
> str(orchis)
'data.frame':          20obs.of      2variables: $flower:num    7 8 6 9 10 11 7 8 10 9...
 $site      : Factor w/ 2 levels "closed","open": 1 1 1 1 1 1 1 1 1 1...
```

7. Carry out a t-test using the formula syntax; you do not need to make assumptions about the variance:

```
> t.test(flower ~ site, data = orchis)
```

8. Now look at the orchis2 data object. It has two columns, flower and site. Use the str() or summary() command to confirm that there are three samples in the sitecolumn:

```
> str(orchis2)
'data.frame':          30obs.of      2variables:
 $flower:num     7 8 6 9 10 11 7 8 10 9...
 $site      : Factor w/ 3 levels "closed","open",..: 1 1 1 1 1 1 1 1 1 1...
```

9. Use a subset instruction to carry out a t-test on the open and closed sites:

```
> t.test(flower ~ site, data = orchis2, subset = site %in% c('open','closed'))
```

10. Now return to the orchid data. Carry out a one-sample test on the opensample to see if it has a mean of less than 7:

```
> t.test(orchid$open, alternative = 'less', mu = 7)

        One Sample t-test

data:    orchid$open
t = -1.9365, df = 9, p-value = 0.04239 alternative
hypothesis: true mean is less than 7
95 percent confidence interval:
       -Inf 6.946615
sample estimates:
mean of x
           6
```

11. Look again at the orchis2 data, which has three samples in the site column. Carry out a t-test on the sprayed sample to see if its mean is greater than 3. You can use either of the followingcommands:

```
> t.test(orchis2$flower[orchis2$site=='sprayed'], mu = 3, alt ='greater')
> with(orchis2, t.test(flower[site=='sprayed'], mu = 3, alt ='g'))


        One Sample t-test


data:    orchis2$flower[orchis2$site=="sprayed"]t =
1.9412, df = 9, p-value =0.04208
alternative hypothesis: true mean is greater than 3
95 percent confidence interval:
 3.061236        Inf
sample estimates:
mean of x
          4.1
```

How It Works

The first part is simply a way to list the data objects by matching items that begin with the text "orc". In the first t-test you had to use the `attach()` command to enable you to specify the column names. Notice that the result begins by telling you that you have carried out the Welch Two-Samplet-test.

In the next case you used the `with()` command to allow R to access the columns in the `orchid` data. By adding `var.equal = TRUE` you carry out the "classic" t-test and treat the variances of the samples as equal. Note that in step 11 you used an abbreviation.

The formula syntax is a convenient way to describe your data; the formula is of the form `response ~ predictor`. The `subset` instruction enables you to select two samples from a column variable; the form of the instruction is `subset = predictor %in% c("item.1","item.2")`.

The Wilcoxon U-Test (Mann- Whitney)

When you have two samples to compare and your data are non-parametric, you can use the U-test. This goes by various names and may be known as the Mann- Whitney U-test or Wilcoxon sign rank test. You use the `wilcox.test()` command to carry out the analysis. You operate this very much like you did when performing the `t.test()` previously.

The `wilcox.test()` command can conduct two-sample or one-sample tests, and you can add a variety of instructions to carry out the test you want. The main options are shown in Table 6-2.

Table 6-2: The wilcox.test() Command and Some of the Options Available.

Command	Explanation
`wilcox.test(sample.1, sample.2)`	Carries out a basic two-sample U-test on the numerical vectors specified.
`mu = 0`	If a one-sample test is carried out, `mu`indicates the value against which the sample should be tested.
`alternative = "two.sided"`	Setsthealternativehypothesis.Thedefaultis`"two.sided"`butyoucanspecify `"greater"` or `"less"`. You can abbreviate the instruction (but you still need quotes).
`conf.int = FALSE`	Sets whether confidence intervals should be reported.
`conf.level = 0.95`	Sets the confidence level of the interval (default = 0.95).
`correct = TRUE`	Bydefaultthecontinuitycorrectionisapplied.Youcanturnthisoffbysetting it to`FALSE`.
`paired = FALSE`	If set to `TRUE`, a matched pair U-test is carried out.
`exact = NULL`	Setswhetheranexactp-valueshouldbecomputed.Thedefaultistodosofor< 50items.
`wilcox.test(y ~ x, data, subset)`	The required data can be specified as a formula of the form `response ~ predictor`.Inthiscasethedatashouldbenamedandasubsetofthepredictor variablecanbespecified.

`subset = predictor %in% c("sample.1", "sample.2")`	If the data is in the form `response~predictor`, the subset instruction can specify which two samples to select from the predictor column of the data.

Two-Sample U-Test

The basic way of using the `wilcox.test()` is to specify the two samples you want to compare as separate vectors, as the following example shows:

```
> data1 ; data2
 [1] 3 5 7 5 3 2 6 8 5 6 9
  [1] 3 5 7 5 3 2 6 8 5 6 9 4 5 7 3 4
> wilcox.test(data1, data2)

        Wilcoxon rank sum test with continuity

correction data:  data1 anddata2
W = 94.5, p-value = 0.7639
alternative hypothesis: true location shift is not equal to 0

Warning message:
In wilcox.test.default(data1, data2) :
  cannot compute exact p-value with
  ties
```

By default the confidence intervals are not calculated and the p-value is adjusted using the "continuity correction"; a message tells you that the latter has been used. In this case you see a warning message because you have tied values in the data. If you set `exact = FALSE`, this message would not be displayed because the p-value would be determined from a normal approximation method.

One-Sample U-Test

If you specify a single numerical vector, a one-sample U-test is carried out; the default is to set `mu = 0`, as in the following example:

```
> wilcox.test(data3, exact = FALSE)

        Wilcoxon signed rank test with continuity correction

data:  data3
V = 78, p-value = 0.002430
alternative hypothesis: true location is not equal to 0
```

In this case the p-value is taken from a normal approximation because the `exact = FALSE` instruction is used. The command has assumed `mu = 0` because it is not specified explicitly.

Using Directional Hypotheses

Both one- and two-sample tests use an alternative hypothesis that the location

shiftisnotequalto0astheirdefault.Thisisessentiallyatwo-sidedhypothesis. You can change this by using the `alternative =` instruction, where you can select `"two.sided"`, `"less"`, or `"greater"` as your alternative hypothesis (an abbreviation is acceptable but you still need quotes, single or double). You can also specify `mu`, the location shift. By default `mu = 0`. In the following example thehypothesisissettosomethingotherthan0:

```
> data3
 [1]  6  7  8  7  6  3  8  910   7  6  9
> summary(data3)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  3.000   6.000   7.000   7.167   8.250  10.000

> wilcox.test(data3, mu = 8, exact = FALSE, conf.int = TRUE, alt =
'less')

        Wilcoxon signed rank test with continuity correction

data:  data3
V = 13.5, p-value = 0.08021
alternative hypothesis: true location is less than 8
95 percent confidence interval:
     -Inf 8.000002
sample estimates:
(pseudo)median
      6.999956
```

In this example a one-sample test is carried out on the `data3` sample vector. The test looks to see if the sample median is less than 8. The instructions also specifytodisplaytheconfidenceintervalandnottouseanexactp-value.

Formula Syntax and Subsetting Samples in the U-test

It is generally a good idea to have your data arranged into a data frame where one column represents the response variable and another represents the predictor variable. In this case you can use the formula syntax to describe the situation and carry out the `wilcox.test()` on your data. This is much the same method you used for the t-test previously. The basic form of the command becomes:

```
wilcox.test(response ~ predictor, data = my.data)
```

You can also use additional instructions as you could with the other syntax. If your predictor variable contains more than two samples, you cannot conduct a U-test and must use a `subset` that contains exactly two samples. The `subset` instruction works like so:

```
wilcox.test(response ~ predictor, data = my.data,
subset = predictor %in% c("sample1", "sample2"))
```

Notice that you use a `c()` command to group the samples together, and their names must be in quotes. The U-test is a useful tool for comparing two samples and is one of the most widely used of all simple statistical tests. Both the `t.test()` and `wilcox.test()`

commands can also deal with matched pair data, which you have not seen yet. This is the subject of the nextsection.

Paired t- and U-Tests

If you have a situation in which you have paired data, you can use matched pair versions of the t-test and the U-test with a simple extra instruction. You simply add `paired = TRUE` as an instruction to your command. It does not matter if the data are in two separate sample columns or are represented as response and predictor as long as you indicate what is required using the appropriate syntax. In fact, R will carry out a paired test even if the data do not really match up as pairs. It is up to you to carry out something sensible. You can use all the regular syntaxandinstructions,soyoucanusesubsettinganddirectionalhypothesesas you like. In the following activity you try a few paired tests for yourself.

Try It Out: Conduct Paired t and U Tests on Some Data

You will need to get the `Beginning.RData` file for this activity: You will require several data objects, which you will use to carry out some paired tests. The file contains all the data you need.

1. Look at the `mpd` data; you can see two samples, `white` and `yellow`. These data are matched pair data and each row represents a bi-colored target. The values are for numbers of whitefly attracted to each half of thetarget.

```
> mpd
  whi  ---,,
1    4     4
2    3     7
3    4     2
4    1     2
5    6     7
6    4    10
7    6     5
8    4     8
```

2. Use a paired U-test (Wilcoxon matched pair test) on these data like so:

```
> wilcox.test(mpd$white, mpd$yellow, exact = FALSE, paired =TRUE)

        Wilcoxon signed rank test with continuity correction

data:  mpd$white andmpd$yellow
V = 6, p-value = 0.2008
alternative hypothesis: true location shift is not equal to0
```

3. Look at the means for the two samples in the mpd data. Round the difference up and then carry out a paired t-test, but set an alternative hypothesisthatthedifferenceinthesemeansislessthanthisdifference:

```
> mean(mpd)
 white
 yellow
 4.000
        5.62
 5
> with(mpd, t.test(white, yellow, paired = TRUE, mu = 2, alt ='less'))
```

```
        Paired  t-test data:

            white andyellow
    t = -3.6958, df = 7, p-value = 0.003849
    alternative hypothesis: true difference in means is less than2
    95 percent confidence interval:
        -Inf 0.2332847
sample estimates:
mean of the differences
                  -1.625
```

4. Look at the `mpd.s` data object. This comprises two columns. One is the response variable `count` and the other is the predictor variable `trap`. These are the same data as the `mpd` and are paired (the only difference is the form of the data object). Carry out a pared t-test on thesedata:

```
> wilcox.test(count ~ trap, data = mpd.s, paired = TRUE, exact =F)

    Wilcoxon signed rank test with continuity correction

data:  count bytrap
V = 6, p-value = 0.2008
alternative hypothesis: true location shift is not equal to0
```

5. Carry out a two-sided and paired t-test on the `mpd.s` data. Set the alternative hypothesis that the difference in means is 1 and show the 99 percent confidenceintervals:

```
> t.test(count ~ trap, data = mpd.s, paired = TRUE, mu = 1, conf.level =0.99)

    Paired t-test

data:  count bytrap
t = -2.6763, df = 7, p-value = 0.03171
alternative hypothesis: true difference in means is not equal to1
99 percent confidence interval:
 -5.057445  1.807445
sample estimates:
mean of the differences
                  -1.625
```

6. Look at the `orchis2` data. Here you have a response variable `flower` and a predictor variable `site`. The predictor variable has three samples (`open`, `closed`, and `sprayed`). Carry out a paired t-test on the `open` and `sprayed`samples:

```
> t.test(flower ~ site, data = orchis2, subset = site %in% c('open', 'sprayed'),
paired = TRUE)

    Paired t-test data:

        flower bysite
t = 4.1461, df = 9, p-value = 0.002499
alternative hypothesis: true difference in means is not equal to0
95 percent confidence interval:
 0.8633494 2.9366506
sample estimates:
mean of the differences
                    1.9
```

How It Works

Simply adding `paired = TRUE` as an instruction to a `t.test()` or

Correlation andCovariance

You can add a variety of additional instructions to these commands. Table 6-3gives a brief summary of them.

Table 6-3: Correlation Commands and Main Options.

Command	Explanation
`cor(x, y = NULL)`	Carriesoutabasiccorrelationbetweenxandy.Ifxisamatrixordataframe,ycanbe omitted.
`cov(x, y = NULL)`	Determines covariance between x and y. If x is a matrix or data frame, y can be omitted.
`cov2cor(V)`	Takes a covariance matrix V and calculates the correlations.
`method =`	Thedefaultis`"pearson"`,but`"spearman"`or`"kendall"`canbespecifiedasthemethods forcorrelationorcovariance.Thesecanbeabbreviatedbutyoustillneedthequotes,and notethattheyarelowercase.
`var(x, y = NULL)`	Determinesthevarianceofx.Ifxisamatrixordataframeoryisspecified,thecovariance is alsodetermined.
`cor.test(x, y)`	Carries out a significance test of the correlation between x and y.
`alternative = "two.sided"`	The default is for a two-sided test but the alternative hypothesis can be given as `"two.sided"`, `"greater"`, or `"less"` and abbreviations are permitted.
`conf.level = 0.95`	If the`method="pearson"`andn>3,theconfidenceintervalswillbeshown.This instructionsettstheconfidencelevelanddefaultsto0.95.
`exact = NULL`	ForKendallorSpearman,shouldanexactp-valuebedetermined?SetthistoTRUEorFALSE (the default NULLis equivalent to FALSE).
`continuity = FALSE`	For Spearman or Kendall tests setting this to `TRUE` carries out a continuity correction.
`cor.test( ~ x + y, data)`	If the data are in a data frame, a formula syntax can be used. This is of the form `~ x + y` wherexandyaretwovariables.Thedataframecanbespecified.Allotherinstructionscan beusedincluding`subset`.
`subset = group %in% "sample"`	Ifthedataincludesagroupingvariable,the`subset`instructioncanbeusedtoselectoneor moresamplesfromthisgrouping.

The commands summarized in Table 6-3enable you to carry out a range of correlation tasks. In the following sections you see a few of these options illustrated, and you can then try some correlations yourself in the activity that follows.

Simple Correlation

Simple correlations are between two continuous variables and you can use the `cor()` command to obtain a correlation coefficient like so:

```
> count = c(9, 25, 15, 2, 14, 25, 24, 47)
```

```
> speed = c(2, 3, 5, 9, 14, 24, 29, 34)

> cor(count,
speed) [1]
0.7237206
```

The default for R is to carry out the Pearson product moment, but you can specify other correlations using the `method =`instruction, like so:

```
> cor(count, speed, method =
'spearman') [1] 0.5269556
```

This example used the Spearman rho correlation but you can also apply Kendall's tau by specifying `method = "kendall"`. Note that you can abbreviate this but you still need the quotes. You also have to uselowercase.

If your vectors are contained within a data frame or some other object, you need to extract them in a different fashion. Look at the `women` data frame. This comes as example data with your distribution ofR.

```
> data(women)
> str(women)
'data.frame':   15obs.of    2variables:
$height:num    58 59 60 61 62 63 64 65 66 67...
 $weight:num    115 117 120 123 126 129 132 135 139 142...
```

You need to use `attach()` or `with()` commands to allow R to "read inside" the data frame and access the variables within. You could also use the `$` syntax so that the command can access the variables as the following example shows:

```
> cor(women$height,
women$weight) [1] 0.9954948
```

In this example the `cor()` command has calculated the Pearson correlation coefficient between the `height` and `weight` variables contained in the `women` data frame.

You can also use the `cor()` command directly on a data frame (or matrix). If you use the data frame `women` that you just looked at, for example, you get the following:

```
> cor(women)
          height
                    weigh
t height
1.00000000.9954948
weight 0.9954948 1.0000000
```

Now you have a correlation matrix that shows you all combinations of the variables in the data frame. When you have more columns the matrix can be much more complex. The following example contains five columns of data:

```
> head(mf)
  Length Speed Algae  NO3 BOD
1     20    12    40 2.25 200
2     21    14    45 2.15 180
3     22    12    45 1.75 135
```

```
4     23    16    80 1.95 120
5     21    20    75 1.95 110
6     20    21    65 2.75 120


> cor(mf)
           Length       Speed      Algae         NO3         BOD
Length  1.000000 -0.34322968  0.7650757  0.45476093          -
Speed        -      1.00000000 -0.1134416  0.02257931   0.198341
Algae    0.765075 -0.11344163  1.0000000  0.37706463          -
NO3      0.454760  0.02257931  0.3770646  1.00000000          -
BOD          -      0.19834122 -0.8365705 -0.37513077   1.000000
```

The correlation matrix can be helpful but you may not always want to see all the possible combinations; indeed, the first column is the response variable and the others are predictor variables. If you choose the `Length` variable and compare it to all the others in the `mf` data frame using the default Pearson coefficient, you can select a single variable and compare it to all the others likeso:

```
> cor(mf$Length, mf)
     Length      Speed     Algae       NO3        BOD
[1,]      1 -0.3432297 0.7650757 0.4547609-0.8055507
```

Covariance

The `cov()` command uses syntax similar to the `cor()` command to examine covariance. The `women` data are used with the `cov()` command in the following example:

```
> cov(women$height, women$weight)
[1] 69
> cov(women)
       height   weight
height     20  69.0000
weight     69240.2095
```

The`cov2cor()`commandisusedtodeterminethecorrelationfromamatrixof covarianceinthefollowingexample:

```
> women.cv = cov(women)
> cov2cor(women.cv)
          height    weight
height 1.00000000.9954948
weight 0.9954948 1.0000000
```

Significance Testing in CorrelationTests

You can apply a significance test to your correlations using the `cor.test()` command. In this case you can compare only two vectors at a time as the following example shows:

```
> cor.test(women$height, women$weight)

        Pearson's product-moment correlation
```

```
data:  women$height andwomen$weight
t = 37.8553, df = 13, p-value = 1.088e-14
alternative hypothesis: true correlation is not equal to 0
95 percent confidence interval:
 0.9860970 0.9985447
sample estimates:
      cor
0.9954948
```

In the previous example you can see that the Pearson correlation has been carried out between `height` and `weight` in the `women` data and the result also shows the statistical significance of the correlation.

Formula Syntax

If your data are contained in a data frame, using the `attach()` or `with()` commands is tedious, as is using the `$` syntax. A formula syntax is available as an alternative, which provides a neater representation of yourdata:

```
> data(cars)
> cor.test(~ speed + dist, data = cars, method = 'spearman',
exact
= F)

       Spearman's rank correlation

rho data:    speed anddist
S = 3532.819, p-value = 8.825e-
14
alternative hypothesis: true rho is not equal to
0 sample estimates:
      rho
0.830356
8
```

Here you examine the `cars` data, which comes built into R. The formula is slightly different from the one that you met previously. Here you specify both variables to the right of the ~. You also give the name of the data as a separate instruction.

All the additional instructions are available when using the formula syntax as wellasthe`subset`instruction.Ifyourdatacontainaseparategroupingcolumn, you can specify the samples to use from it using an instruction along the followinglines:

```
subset = grouping %in% "sample"
```

Correlation is a common method used widely in many areas of study. In the following activity you will be able to practice carrying out correlationand covariance of some data.

Try It Out: Carry Out Correlation andCovariance

Use the fw, fw2, and fw3 data from the Beginning.RData file for this activity. The other data items are built into R.

1. Look at the fw data object; this contains two columns, countand speed. Conduct a Pearson correlation on these two variables:

```
> cor(fw$count, fw$speed)
[1] 0.7237206
```

2. Now look at the swissdata object; this is built into R. Use Kendall's taucorrelationtocreateamatrixofcorrelations:

```
> cor(swiss, method = 'kendall')
```

3. The swiss data produced a sizeable matrix. Simplify this by looking at the Fertility variable and correlating that to the others in the dataset. This time use the Spearman rhocorrelation.

```
> cor(swiss$Fertility, swiss, method ='spearman')
      FertilityAgricultureExamination      Education
                CatholicInfant.Mortality[1,] 1            0.2426643       -
0.660903-0.44325770.4136456              0.4371367
```

4. Now look at the fw data object. It has two variables, count and speed. Create a covariancematrix:

```
> (fw.cov = cov(fw))
          count
                    spee
d count
185.8393123.0000
speed 123.0000 155.4286
```

5. Convertthecovariancematrixintoacorrelation:

```
> cov2cor(fw.cov)
           count
                     spee
d count
1.00000000.7237206
speed 0.7237206 1.0000000
```

6. Look at the fw2 data object. This has the same number of rows as the fw object. It also has two columns, abund and flow. Carry out a correlation between the columns of one data frame and theother:

```
> cor(fw, fw2)
           abund
                      flo
w count
0.99057590.7066437
speed 0.6527244 0.9889997
```

7. Carry out a Spearman rho test of significance on the count andspeed variables from the fwdata:

```
> with(fw, cor.test(count, speed, method ='spearman'))
  Spearman's rank correlation rho
```

```
data:  count andspeed
S = 39.7357, p-value = 0.1796
alternative hypothesis: true rho is not equal to 0 sample
estimates:
      rho
0.5269556


Warning message:
In cor.test.default(count, speed, method = "spearman") :
  Cannot compute exact p-values withties
```

8. Now look at the fw2 data again. Conduct a Pearson correlation between the abund and flow variables. Set the confidence intervals to the 99 percent level and use an alternative hypothesis that the correlation is greater than 0:

```
> cor.test(fw2$abund, fw2$flow, conf = 0.99, alt ='greater')

      Pearson's product-moment correlation

data:  fw2$abund andfw2$flow
t = 2.0738, df = 6, p-value =0.04173
alternative hypothesis: true correlation is greater than0
99 percent confidence interval:
 -0.265223  1.000000
sample estimates:
      cor
0.6461473
```

9. Use the formula syntax to carry out a Kendall tau correlation significancetestbetweentheLengthandNO3variablesfromthemfdata object:

```
> cor.test(~ Length + NO3, data = mf, method = 'k', exact =F)

    Kendall's rank correlation tau data:

        Length andNO3
z = 1.969, p-value = 0.04895
alternative hypothesis: true tau is not equal to 0 sample
estimates:
      tau
0.2959383
```

10. Lookatthefw3dataobject.Thisisthesameasfw,exceptthatthere isanadditionalgroupingvariablecalledcover.Useasubsetofthedata that corresponds to the open group and carry out a Pearson correlation significancetest:

```
> cor.test(~ count + speed, data = fw3, subset = cover %in%'open')

    Pearson's product-moment correlation

data:  count andspeed
t = -1.1225, df = 2, p-value =0.3783
alternative hypothesis: true correlation is not equal to0
95 percent confidence interval:
```

-0.9907848 0.8432203

Tests for Association

When you have categorical data you can look for associations between categories by using the chi-squared test. Routines to achieve this are accessed using the `chisq.test()` command. You can add various additional instructions to the basic command to suit your requirements. These are summarized in Table 6-4.

Table 6-4: The Chi-Squared Test and its Various Options.

Command	Explanation
`chisq.test(x, y = NULL)`	A basic chi-squared test is carried out on a matrix or dataframe. If x is provided as a vector, a second vector can be supplied. If x is a single vector and y is not given, a goodness of fit test is carried out.
`correct =`	If the data form a 2 × 2 contingency table the Yates' correction is applied.
`p =`	A vector of probabilities for use with a goodness of fit test. If p is not given, the goodness of fit test is that the probabilities are all equal.
`rescale.p =`	If `TRUE`, p is rescaled to sum to 1. For use with goodness of fit tests.
`simulate.p.value`	If set to `TRUE`, a Monte Carlo simulation is used to calculate p-values.
`B = 2000`	The number of replicates to use in the Monte Carlo simulation.

Multiple Categories: Chi-Squared Tests

The most common use for a chi-squared test is where you have multiple categories and want to see if associations exist between them. In the following example you can see some categorical data set out in a data frame. You have seen these data before:

```
> bird.df
             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            46       16        8       4        0
Robin             9        3        0       0        2
Song Thrush       4        0        6       0        0
```

The data here are already in a contingency table and each cell represents a unique combination of the two categories; here you have several habitats and several species. You run the `chisq.test()` command simply by giving the name of the data to the command like so:

```
> bird.cs =
chisq.test(bird.df) Warning
message:
In chisq.test(bird.df) : Chi-squared approximation may
be incorrect
> bird.cs

        Pearson's Chi-squared
```

```
test data:   bird.df
X-squared = 78.2736, df = 20, p-value = 7.694e-09
```

In this case you give the result a name and set it up as a new object, which you examine in more detail in a moment. You get an error message in this example; this is because you have some small values for your observed data and the expected values will probably include some that are smaller than 5. When you issue the name of the result object you see a very brief result that contains the salientpoints.

Your original data were in the form of a data frame but you might also have used a matrix. If that were so, the result is exactly the same. You can also use a table result; perhaps the result of using the `xtabs()` command on the raw data. In any event you end up with a result object, which you can examine in more detail. You might start by trying a `summary()` command:

```
> summary(bird.cs)
          LengthClass   Mode
statistic  1     -none-numeric
parameter  1     -none-numeric
p.value    1     -none-numeric
method     1     -none-character
data.name  1     -none-character
observed  30     -none-numeric
expected  30     -none-numeric
residuals30      -none-numeric
```

This does not produce the result that you may have expected. However, it does show that the result object you created contains several parts. A simpler way to see what you are dealing with is to use the `names()` command:

```
> names(bird.cs)
[1] "statistic""parameter""p.value"    "method"
                                       "data.name" "observed"
[7]"expected"   "residuals"
```

You can access the various parts of your result object by using the `$` syntax and adding the part you want to examine. For example:

```
> bird.cs$stat
X-square
 d
 78.2736
 4
> bird.cs$p.val
[1] 7.693581e-
09
```

Here you select the statistic (the X^2 value) and the p-value; notice that you do not need to use the full name here, an abbreviation is fine as long as it is unambiguous. You can see the calculated expected values as well as the Pearson residuals by using the appropriate abbreviation. In the following example you look at the expectedvalues:

```
> bird.cs$exp
                Garden  Hedgero  Parkland   Pasture  Woodland
Blackbird    59.915254 10.95593 23.623729 4.4508475 2.0542373
Chaffinch    17.203390  3.14576  6.783051 1.2779661 0.5898305
Great Tit    39.745763  7.26779 15.671186 2.9525424 1.3627119
House        43.898305  8.02711 17.308475 3.2610169 1.5050847
Robin         8.305085  1.51864  3.274576 0.6169492 0.2847458
Song Thrush   5.932203  1.08474  2.338983 0.4406780 0.2033898
```

You can see in this example that you have some expected values < 5 and this is the reason for the warning message. You might prefer to display the values as whole numbers and you can adjust the output "on the fly" by using the `round()` command to choose how many decimal points to display the values likeso:

```
                    > 0)
  round(bird.cs$exp, Hedgerow Parkland Pasture Woodland
Blackbird          60       11       24       4        2
Chaffinch          17        3        7       1        1
GreatTit           40        7       16       3        1
HouseSparrow       44        8       17       3        2
Robin               8        2        3       1        0
SongThrush          6        1        2       0        0
```

In this instance you chose to use no decimals at all and so use 0 as an instruction in the `round()` command.

Monte Carlo Simulation

You can decide to determine the p-value by a slightly different method and can use a Monte Carlo simulation to do this. You add an extra instruction to the `chisq.test()` command, `simulate.p.value = TRUE`, like so:

```
> chisq.test(bird.df, simulate.p.value = TRUE, B = 2500)
```

```
2500Pearson's Chi-squared test with simulated p-value (based on
replicates)

data:  bird.df
X-squared = 78.2736, df = NA, p-value = 0.0003998
```

The default is that `simulate.p.value = FALSE` and that `B = 2000`. The latter is the number of replicates to use in the Monte Carlo test, which is set to 2500 for thisexample.

Yates' Correction for 2 × 2 Tables

When you have a 2 × 2 contingency table it is common to apply the Yates' correction. By default this is used if the contingency table has two rows and two columns. You can turn off the correction using the `correct = FALSE` instruction in the command. In the following example you can see a 2 × 2 table:

```
> nd
          Urt.dio.y Urt.dio.n
Rum.obt.y        96        41
Rum.obt.n        26        57

> chisq.test(nd)

       Pearson's Chi-squared test with Yates'
continuity correction

data:  nd
X-squared = 29.8653, df = 1, p-value = 4.631e-08

> chisq.test(nd, correct =
       FALSE)

       Pearson's Chi-squared

test data:   nd
X-squared = 31.4143, df = 1, p-value = 2.084e-08
```

At the top you see the data and when you run the `chisq.test()` command you see that Yates' correction is applied automatically. In the second example you force the command not to apply the correction by setting `correct = FALSE`. Yates' correction is applied only when the matrix is 2 × 2, and even if you tell R to apply the correction explicitly it will do so only if the table is 2 ×2.

Single Category: Goodness of FitTests

In the following example you have a simple data frame containing two columns; the first column contains values relating to an old survey. The second column contains values relating to a new survey. You want to see if the proportions of the new survey match the old one, so you

perform a goodness of fittest:

```
> survey
        old new
woody    23  19
shrubby  34  30
tall    132 111
short    98 101
grassy   45  52
mossy    53  26
```

To run the test you use the `chisq.test()` command, but this time you must specify the test data as a single vector and also point to the vector that contains the probabilities:

```
> survey.cs = chisq.test(survey$new, p = survey$old,
rescale.p = TRUE)
> survey.cs

        Chi-squared test for given

probabilities data:    survey$new
X-squared = 15.8389, df = 5, p-value = 0.00732
```

In this example you did not have the probabilities as true probabilities but as frequencies; you use the `rescale.p = TRUE` instruction to make sure that these are converted to probabilities (this instruction is set to `FALSE`by default).

The result contains all the usual items for a chi-squared result object, but if you display the expected values, for example, you do not automatically get to see the row names, even though they are present in the data:

```
> survey.cs$exp
[1]  20.25195  29.93766116.22857   86.29091  39.62338  46.66753
```

You cangettherownamesfromtheoriginaldatausingthe`row.names()` command. You could set the names of the expected values in the following way:

```
names(survey.cs$expected) = row.names(survey)
> survey.cs$exp
    woody   shrubby      tall     short    grassy     mossy
 20.25195  29.93766116.22857   86.29091  39.62338  46.66753
```

You could do something similar for the residuals and then when you inspected your result it would be easier to keep track of which value was related to which category.

In the following activity you can get a chance to practice the chi-squared test for association as well as goodness of fit by using a simple data example.

Try It Out: Carry Out Chi-Squared Tests on Some Data

Use the `bees` data object from the `Beginning.RData` file for this activity, which you will use to carry out a range of association and goodness of fit tests. The data are in a data frame and represent visits by various bee species to different plant species.

1. Carry out a basic chi-squared test on these data and save the result as a namedobject:

```
> bees
            Buff.t Garden. Red.ta Honey. Carder.
Thistle         10       8     18     12       8
Vipers.bug       1       3      9     13      27
Golden.rai      37      19      1     16       6
Yellow.alf       5       6      2      9      32
Blackberry      12       4      4     10      23

> (bees.cs = chisq.test(bees))

   Pearson's Chi-

squared test data:

     bees
X-squared = 120.6531, df = 16, p-value <2.2e-16
```

2. Look at the result you just obtained—it contains several parts. DisplaythePearsonresidualsfortheresult:

```
> names(bees.cs)
[1] "statistic""parameter""p.value"   "method"
                                "data.name""observed""ex

pected"
[8] "residuals"
> bees.cs$resid
            Buff.tai Garden.  Red.ta Honey.be Carder.
Thistle           - 0.14762 4.5446 0.180797       -
Vipers.bug        -       - 1.1699 0.676264 2.34830
Golden.rai 4.696200 2.53235      -        -       -
Yellow.alf        -       -      -        - 3.44158
Blackberry 0.094236       -      -        - 1.38515
```

3. Now run the chi-squared test again but this time use a Monte Carlo simulation with 3000 replicates to determine the p-value:

```
> (bees.cs = chisq.test(bees, simulate.p.value = TRUE, B
=3000))

   Pearson's Chi-squared test with simulated p-value
(based on 3000 replicates)

data:bees
X-squared = 120.6531, df = NA, p-value =0.0003332
```

4. Look at a portion of the data as a 2 × 2 contingency table. Examine the effect of Yates' correction on this subset:

```
> bees[1:2, 4:5]
         Honey.bee Carder.bee
Thistle         12         8
Vipers.bugloss  13        27

> chisq.test(bees[1:2, 4:5], correct =FALSE)

   Pearson's Chi-
```

```
squared test data:

     bees[1:2,4:5]
X-squared = 4.1486, df = 1, p-value =0.04167

> chisq.test(bees[1:2, 4:5], correct =TRUE)

   Pearson's Chi-squared test with Yates'

continuity correction data:   bees[1:2,4:5]
X-squared = 3.0943, df = 1, p-value =0.07857
```

5. Look at the last two columns, representing two bee species. Carry out a goodness of fit test to determine if the proportions of visits are the same:

```
> with(bees, chisq.test(Honey.bee, p = Carder.bee, rescale
=T))

   Chi-squared test for given

probabilities data: Honey.bee
X-squared = 58.088, df = 4, p-value =7.313e-12

Warning message:
In chisq.test(Honey.bee, p = Carder.bee,
  rescale = T) : Chi-squared approximation
  may beincorrect
```

6. Carry out the same goodness of fit test but use a simulation to determinethep-value(youcanabbreviatethecommand):

```
> with(bees, chisq.test(Honey.bee, p = Carder.bee, rescale =
T, sim =T))

   Chi-squared test for given probabilities with
simulated p-value (based on 2000
   replicates)

data:Honey.bee
X-squared = 58.088, df = NA, p-value =0.0004998
```

7. Nowlookatasinglecolumnandcarryoutagoodnessoffittest.This timeomitthep=instructiontotestthefittoequalprobabilities:

```
> chisq.test(bees$Honey.bee)
    Chi-squared test for givenprobabilities

 data:bees$Honey.bee
 X-squared = 2.5, df = 4, p-value =0.6446
```

How It Works

The basic form of the `chisq.test()` command will operate on a matrix or data frame. By enclosing the entire command in parentheses you can get the result object to display immediately. The results of many commands are stored as a list containing

several elements, and you can see what is available using the `names()` command and view them using the `$` syntax.

The p-value can be determined using a Monte Carlo simulation by using the `simulate.p.value` and `B` instructions. If the data form a 2 × 2 contingency, then Yates' correction is automatically applied but only if the Monte Carlo simulation is *not* used.

To conduct a goodness of fit test you must specify `p`, the vector of probabilities; if this does not sum to 1 you will get an error unless you use `rescale.p = TRUE`. You can use a Monte Carlo simulation on a goodness of fit test. If a single vector is specified, a goodness of fit test is carried out but the probabilities are assumed to be equal.

Summary

- A variety of simple statistical tests are built into R.
- The t-test can be carried out using the `t.test()` command. This can conduct one- or two-sample tests and a range of options allow one-tailed and two- tailed tests.
- The U-test is accessed via the `wilcox.test()` command. This non- parametric test of differences can be applied as one-sample or two-sample versions.
- Matched paired data can be analyzed using t-test or U-test by the simple addition of the `paired = TRUE` instruction in the `t.test()` or `wilcox.test()` commands.
- The `subset` instruction can be used to select one or more samples from a variable containing several groups.
- Correlation and covariance can be carried out on pairs of vectors, or on entire dataframes or matrix objects using the `cor()` and `cov()` commands.

  A single variable can be specified to produce a targeted correlation or covariance matrix.
- Three types of correlation can be used; Pearson's Product Moment, Spearman's rho or Kendall's tau.
- Correlation hypothesis tests can be carried out using Pearson, Spearman, or Kendall methods via the `cor.test()` command. Two variables can be specified as separate vectors or using the formula syntax.
- Tests using categorical data can be carried out via the `chisq.test()` command. This can conduct standard tests of association (chi-squared tests) or goodness of fit tests. Monte Carlo simulation can be used to produce the p-value.

Exercises

You can find answers to these exercises in Appendix A.

Use the `hogl` and `bv` data objects in the `Beginning.RData` file for these exercises. The `sleep`, `InsectSprays`, and `mtcars` data objects are part of the regular distribution of R.

1. Look at the `InsectSprays` data. Compare the effectiveness of spray types A and B using a t-test.

2. Look at the `hogl` data. This data frame contains two columns, representing the abundance of a freshwater invertebrate (`hoglouse`) at two habitats (`slow` and `fast`). Use a U-test to compare the abundance.

3. Look at the `sleep` data; you will see that it has three columns. The `extra` column represents time of additional sleep induced by a drug. The `group` column gives a numeric value; this is the drug (1 or 2). The final column, `ID`, is simply the patient identification. Each patient was given both drugs (on different occasions) and the time of additional sleep recorded. Carry out a paired t-test on the additional sleep times and the different drugs.

4. Look at the `mtcars` data that gives data on the fuel consumption and other features of some automobiles from the 1970s. First look at a correlation matrix of these data, then focus on the correlation between `mpg` and the other variables. Finally, carry out a correlation test on the `mpg` and `qsec`(time taken to travel a quarter mile) variables.

5. Look at the `bv` data. Here you can see a column, `visit`, which relates to numbers of bees visiting various colors of flowers. The `ratio` column refers to the relative numbers of visits from a previous experiment. Carry out a goodness of fit test to see if the two experiments have given the same results.

What You Learned in This Chapter

Topic	Key Points
T-test: `t.test(data1, data2 =`	Student'st-testcanbecarriedoutusingthe`t.test()`command.Youmustspecify two vectors if you want a two-sample test; otherwise a one-sample test is conducted.Aformulacanbespecifiedifthedataareintheappropriatelayout
U-test: `wilcox.test(data1, data2 = NULL)wilcox.t`	The U-test (Mann-Whitney or Wilcoxon test) can be carried out using the `wilcox.test()`command.One-sampleortwo-sampletestscanbeexecutedanda formula can be used if the data are in an appropriate layout.You can use various
Paired tests: `t.test(x, y, paired =`	PairedversionsofthetandtheU-testcanbecarriedoutbyaddingthe`paired =TRUE`instructiontothecommand.PairscontainingNAitemsaredropped.You
Subsetting: `subset = group %in%`	Ifyourdataareinaformwhereyouhavearesponsevariableandapredictor variableyoucanselectasubsetofthedatausingthe`subset`instruction.
Covariance: `cov(x, y)` Pearson, Spearman, Kendall	Covariance can be examined using the `cov()` command. You can specify two objects,whichcanbevector,dataframe,ormatrix.Allobjectsmustbeofequal length. You can specify one of `"pearson"` (default), `"spearman"` or `"kendall"`
Correlation: `cor(x, y)` Pearson, Spearman,	Correlationcanbecarriedoutusingthe`cor()`command.Youcanspecifytwo objects,whichcanbevector,dataframe,ormatrix.Allobjectsmustbeofequal

Correlation hypothesis tests: **cor.test(x, y)cor.test(~**	Correlationhypothesistestscanbecarriedoutusingthe`cor.test()` command. You can specify two vectors or use the formula syntax. Unlike `cov()` or `cor()` commandsyoucancompareonlytwovariablesatatime.Youcanspecifyone of `"pearson"` (default), `"spearman"`, or `"kendall"` (can be
Association tests: **chisq.test(x, y = NULL)**	Chi-squared tests of association can be carried out using the `chisq.test()` command.If`x`isadataframeormatrix,`y`isignored.Yates'correctionisappli
Goodness of fit tests: **chisq.test(x,**	Chi-squared goodness of fit tests can be carried out using the `chisq.test()` command.Asinglevectormustbegivenforthetestdataandtheprobabilitie sto test against are given as p. If they do not sum to 1, you can use
Monte Carlo simulation: **simulate.p.va**	Forchi-squaredtestsofassociationorgoodnessoffityoucandeterminethep-valuebyMonteCarlosimulationusingthe`simulate.p.value`instr
Rounding values: **round(object, digits = 6)**	The level of precision of displayed results can be altered using the `round()` command.Youspecifythenumericalresultstouseandthenumberofdigit

UNIT-III

Terminology

Topic	Key Points
Simple regression: cor.test() lm()	Simplelinearregressionthatcouldbecarriedoutusingthecor.test()commandca n also be carried out using the lm() command. The lm() command is more powerful andflexible.
Regression coef() fitted() resid()confint()	Results objects created using the lm()command contain information that extracted using the $syntax, but also using dedicated commands. You can also obtain
Best-fit lines: abline()	Youcanaddlinesofbest-fitusingtheabline()commandiftheyarestraightlines. The command can determine the slope and intercept from the result of an lm() command.
ANOVA and **lm()**: anova()	Analysisandlinearmodelingareverysimilar,andinmanycasesyoucancarryout an ANOVA using the lm() command. The anova() command produces the classic ANOVAtablefromtheresultofanlm()command.
Linear modeling:formul a syntax	Thebasisofthelm()commandistheformulasyntax(alsoknownasmodelsyntax) . This takes the form of response ~ predictor(s). Complex models can be specified using thissyntax.
Model building: add1()drop1()	Youcanbuildregressionmodelsinaforwardstepwisemannerorbyusingbackwa rd deletion. Moving forward, terms are added using the add1() command. Backward deletionusesthedrop1()command.
Comparing regression models:	You can compare regression models using the anova()command.
Curvilinear regressionlm()	Regressionmodelsdonothavetobeintheformofastraightline,andaslongasthe relationshipcanbedescribedmathematically,theregressioncanbedescribedu sing themodelsyntaxandcarriedoutwiththelm()command.
Adding best-fit lines: abline()fitted() spline()	Lines of best-fit can be added using the abline() command if they are straight. If theyarecurvilinear,thelines()commandisused.Thelinescanbecurvedusingth e
Confidence intervals: predict()lines() spline()	Confidence intervals can be determined on the fit of an lm() model using the predict()command.Thesecanthenbeplottedonaregressiongraphusingthe lines()command;usethespline()commandtoproduceasmoothcurveif necessary.
Diagnostic plots:	Youcanusetheplot()commandonanlm()resultobjecttoproducediagnostic plots.

Chapter 7

Introduction to Graphical Analysis

What you will learn in this chapter:

- How to create a range of graphs to summarize your data and results How to create
- box-whisker plots
- How to create scatter plots, including multiple correlation plots How to
- create line graphs
- How to create pie charts How
- to create bar charts
- How to move graphs from R to other programs and save graphs as files on disk

Graphs are a powerful way to present your data and results in a concise manner. Whatever kind of data you have, there is a way to illustrate it graphically. A graph is more readily understandable than words and numbers, and producing good graphs is a vital skill. Some graphs are also useful in examining data so that you can gain some idea of patterns that may exist; this candirectyoutowardthecorrectstatisticalanalysis.

R has powerful and flexible graphical capabilities. In general terms, R has two kinds of graphical commands: some commands generate a basic plot of some sort, and other commands are used to tweak the output and to produce a more customizedfinish.

You have already encountered some graphical commands in previous chapters. This chapter focuses on some of the basic graph types that you may typically need to create. In Chapter 11, you will revisit the graphical commands and add a variety of extras to lift your graphs from the merely adequate, to fully polished publication qualitymaterial.

Box-whisker Plots

The box-whisker plot (often abbreviated to boxplot) is a useful way to visualize complex data where you have multiple samples. In general, you are looking to display differences between samples. The basic form of the box-whisker plot shows the median value, the quartiles (or hinges), and the max/min values. This means that you get a lot of information in a compact manner. The box-whisker plot is also useful to visualize a single sample because you can show outliers if you choose. You can use the `boxplot()` command to create box-whisker plots. The command can work in a variety of ways to visualize simple or quite complexdata.

Basic Boxplots

The following example shows a simple data frame composed of two columns:

```
> 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
```

You have seen these data before. You can use the `boxplot()` command to visualize one of the variables here:

```
> boxplot(fw$speed)
```

This produces a simple graph like Figure 7-1. This graph shows the typical layout of a box-whisker plot. The stripe shows the median, the box represents the upper and lower hinges, and the whiskers show the maximum and minimum values.

If you have several items to plot, you can simply give the vector names in the `boxplot()` command:

```
> boxplot(fw$count, fw$speed)
```

The resulting graph appears like Figure 7-2. In this case you specify vectors that correspond to the two columns in the data frame, but they could be completelyseparate.

Figure 7-1:

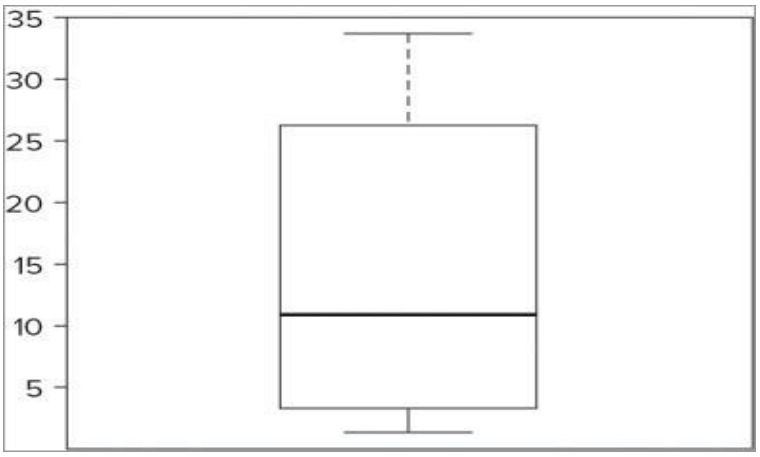
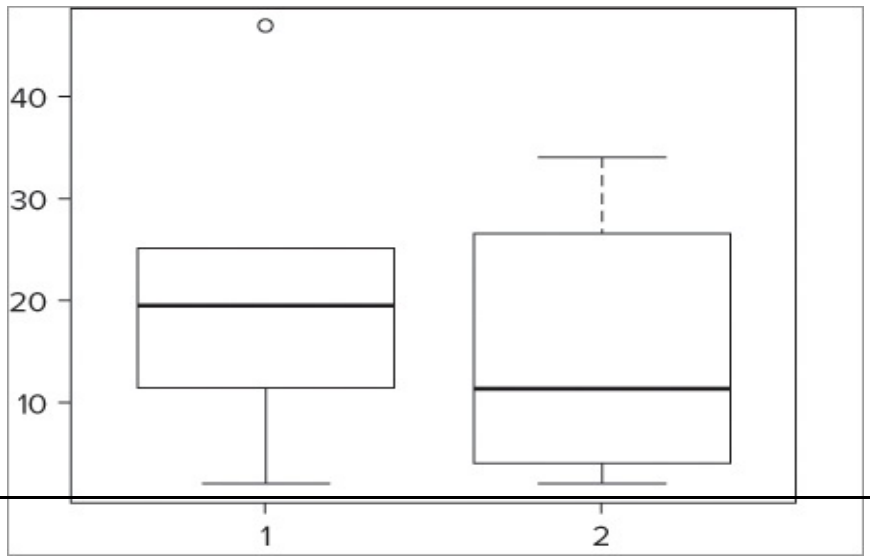


Figure 7-2:



Customizing Boxplots

A plot without labels is useless; the plot needs labels. You can use the `xlab` and `ylab` instructions to label the axes. You can use the `names` instruction to set the labels (currently displayed as 1 and 2) for the two samples, like so:

```
> boxplot(fw$count, fw$speed, names = c('count', 'speed'))
> title(xlab = 'Variable', ylab = 'Value')
```

The resulting plot looks like Figure 7-3. In this case you used the `title()` commandtoaddtheaxislabels,butyoucouldhavespecified`xlab` and`ylab` within the `boxplot()` command.

Now you have names for each of the samples as well as axis labels. Notice that the whiskers of the `count` sample do not extend to the top, and that you appear to have a separate point displayed. You can determine how far out the whiskers extend, but by default this is 1.5 times the interquartile range. You can alter this by using the `range =` instruction; if you specify `range = 0` as shown in the followingexample,thewhiskersextendtothemaximumandminimumvalues:

```
> boxplot(fw$count, fw$speed, names = c('count', 'speed'),
range = 0,
xlab = 'Variable', ylab = 'Value', col = 'gray90')
```

The final graph appears like Figure 7-4. Here you not only force the whiskers to extend to the full max and min values, but you also set the box colors to a light gray. You can see which colors are available using the `colors()` command.

Figure 7-3:

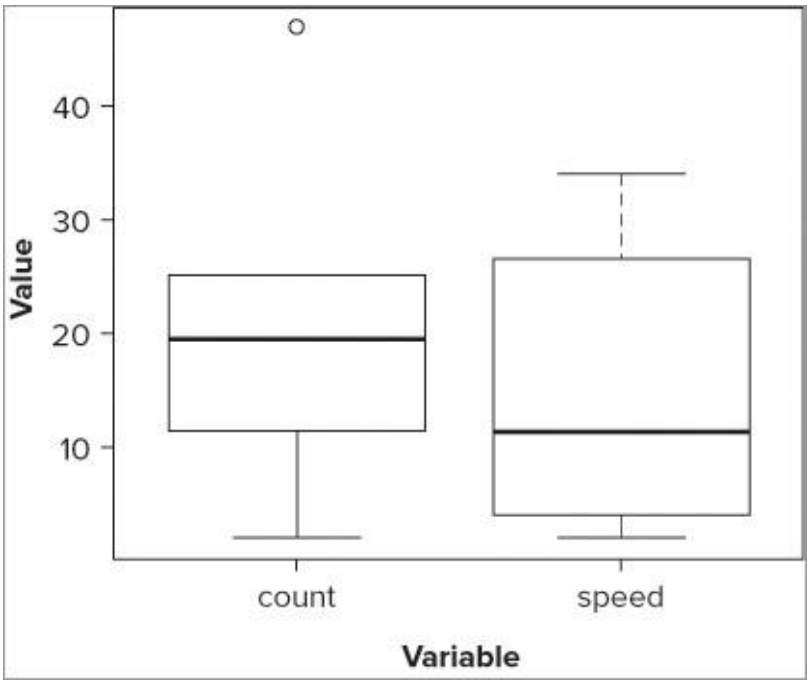
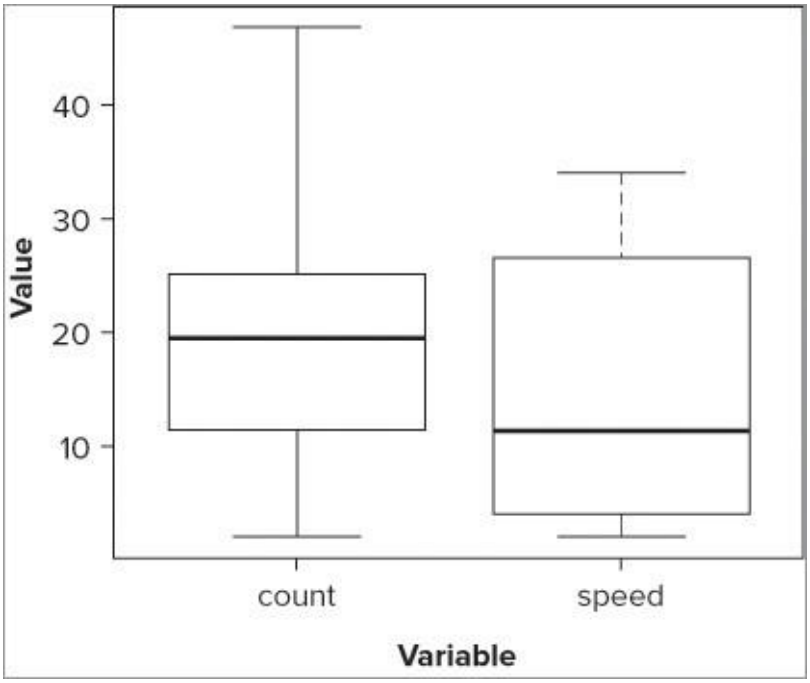


Figure 7-4:



In the examples you have seen so far the data samples being plotted are separate numerical vectors. You will often have your data in a different arrangement; commonly you have a data frame with one column representing the response variable and another representing a predictor (or grouping) variable. In practice this means you have one vector containing all the numerical data and another vector containing the grouping information as text. Look at the followingexample:

```
> gras
  rich  graz
1   12   mow
2   15   mow
3   17   mow
4   11   mow
5   15   mow
6    8  unmo
7    9  unmo
8    7  unmo
9    9  unmo
```

With data in this format, it is best to use the same formula notation you used with `t.test()`. When doing so, you use the `~` symbol to separate the response variable to the left and the predictor (grouping) variable to the right. You can also instruct the command where to find the data and set `range = 0` to force the whiskers to the maximum and minimum as before. See the following example fordetails:

```
> boxplot(rich ~ graze, data = grass, range = 0)
> title(xlab = 'cutting treatment', ylab = 'species richness')
```

Here you also chose to add the axis labels separately with the `title()` command. Notice

this time that the samples are automatically labeled; the command takes the names of the samples from the levels of the factor, presented in alphabetical order. The resulting graph looks like Figure 7-5.

Figure 7-5:

You can give additional instructions to the command; these are listed in the help entry for the `boxplot()` command. Before you learn those, however, first take a look at one additional option: horizontalbars.

Horizontal Boxplots

With a simple additional instruction you can display the bars horizontally rather than vertically (which is the default):

```
> boxplot(rich ~ graze, data = grass, range = 0,
horizontal = TRUE)
> title(ylab = 'cutting treatment', xlab = 'species richness')
```

When you use the `horizontal = TRUE` instruction, your graph is displayed with horizontal bars (see Figure 7-6). Notice how with the `title()` command you had to switch the `x` and `y` labels. The `xlab` instruction refers to the horizontal axis and the `ylab` instruction refers to thevertical.

In the following activity you can practice creating box-whisker plots using some data in various forms.

Figure 7-6:

Figure 7-7:

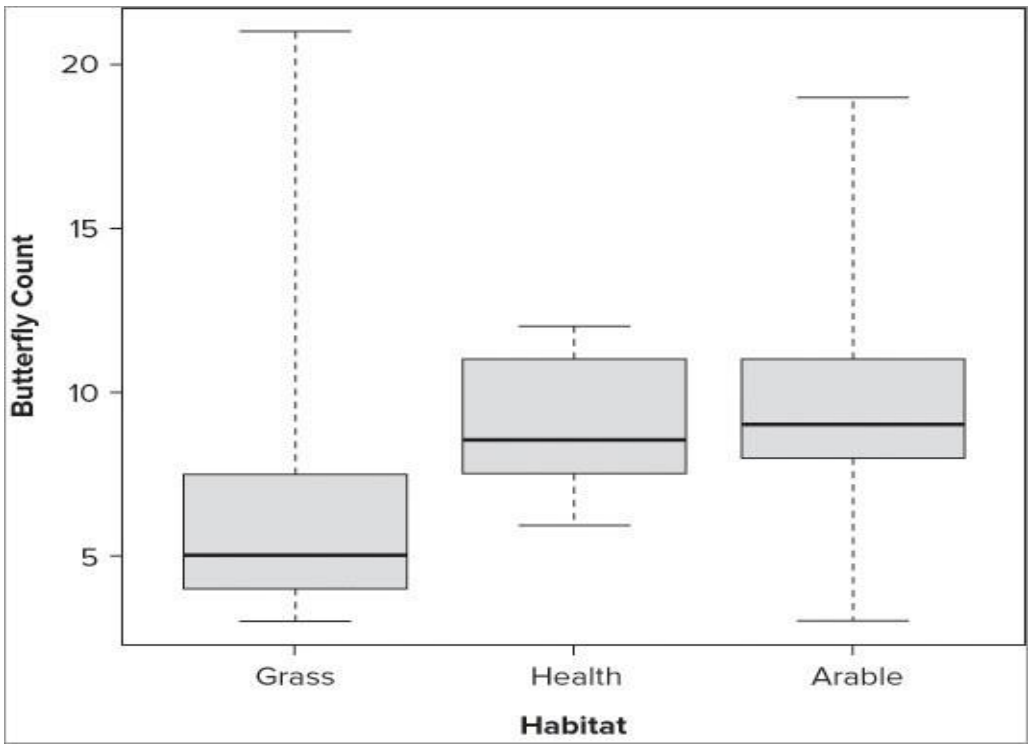
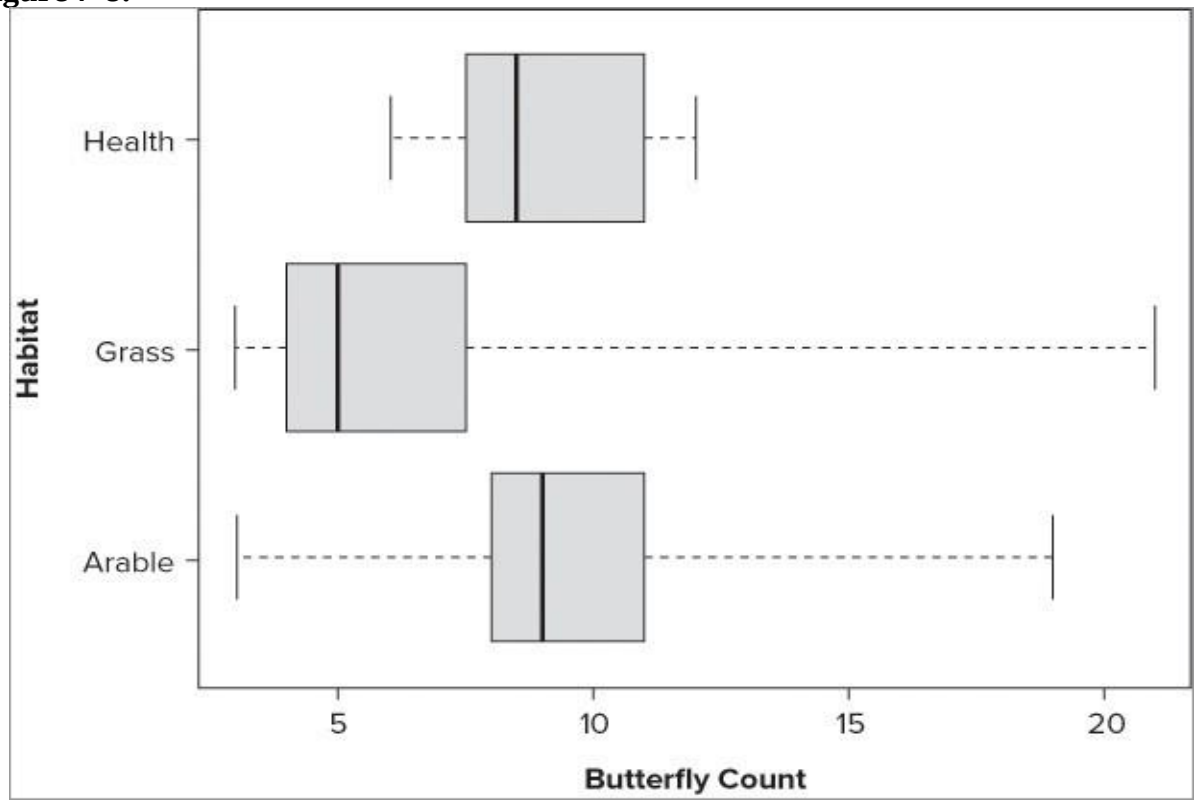


Figure 7-8:



If your data are in the form of a response variable and a predictor (grouping) variable, the samples will be in alphabetical order.

You can reorder the samples in a simple data frame by specifying them explicitly. Forexample:

```
> names(bf)
[1]"Grass"   "Heath"  "Arable"
> boxplot(bf[c(2,3,1)])
> boxplot(bf[c('Heath', 'Arable', 'Grass')])
```

The two `boxplot()` commands produce the graph with the samples in a new order.

If your data are in a `response ~ grouping` layout, it is harder to reorder the graph. The following example shows how you might achieve a reordering:

```
> with(bfs, boxplot(count[site=='Heath'],
count[site=='Arable'], count[site=='Grass'], names =
c('Heath', 'Arable', 'Grass'))) # data frame
```

The box-whisker plot is very useful because it conveys a lot of information in a compact manner. R is able to produce this type of plot easily. In the rest of this chapter you see some of the other graphs that R is able to produce.

Scatter Plots

The basic `plot()` command is an example of a generic function that can be pressed into service for a variety of uses. Many specialized statistical routines include a plotting routine to produce a specialized graph. For the time being, however, you will use the `plot()` command to produce `xy` scatter plots. The scatter plot is used especially to show the relationship between two variables. You met a version of this in Chapter 5 when you looked at QQ plots and the normaldistribution.

Basic Scatter Plots

Thefollowingdataframecontainstwocolumnsofnumericvalues,andbecause they contain the same number of observations, they could form the basis for a scatterplot:

```
> 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
```

The basic form of the `plot()` command requires you to specify the `x` and `y` data, each being a numeric vector. You use it like so:

```
plot(x, y, ...)
```

If you have your data contained in a data frame as in the following example, you must use the `$` syntax to get at the variables; you might also use the `with()` or `attach()` commands. For the example data here, the following commands all produce a similarresult:

```
> plot(fw$speed, fw$count)
> with(fw, plot(speed, count))

> attach(fw)
> plot(speed, count)
> detach(fw)
```

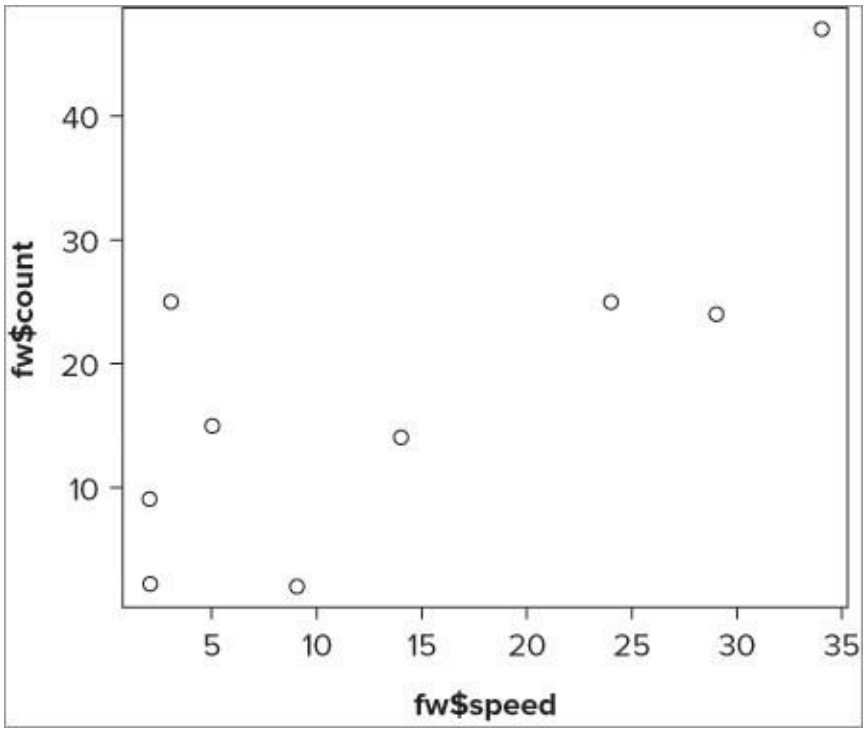
The resulting graph looks like Figure 7-9. Notice that the names of the axis labels match up with what you typed into the command. In this case you used the `$`syntax to extract the variables; these are reflected in the labels.

Adding Axis Labels

You can produce your own axis labels easily using the `xlab` and `ylab` instructions. For example, to create labels for these data you might use something like thefollowing:

```
> plot(fw$speed, fw$count, xlab = 'Speed m/s', ylab = 'Count
of Mayfly')
```

Figure 7-9:



Previously you used the `title()` command to add axis titles. If you try this here you end up writing text over the top of the existing title. You can still use the `title()` command to add axis titles later, but you need to produce blank titles to start with. You must set each title in

the `plot()` command to blank using
a pair of quotes as shown in the following:

```
> plot(fw$speed, fw$count, xlab = "", ylab = "")
```

This is quite convoluted so most of the time it's better to set the titles as part of the `plot()` command at the outset.

Plotting Symbols

You can use many other graphical parameters to modify your basic scatter plot. You might want to alter the plotting symbol, for example. This is useful if you want to add more points to your graph later. The `pch =` instruction refers to the **plotting character**, and can be specified in one of several ways. You can type an integer value and this code will be reflected in the symbol/character produced. For values from 0 to 25, you get symbols that look like the ones depicted in Figure7-10.

Figure 7-10:

These were produced on a scatter plot using the following lines of command:

```
> plot(0:25, rep(1, 26), pch = 0:25, cex = 2)
> text(0:25, 0.95, as.character(0:25))
```

The first part produces a series of points, and sets the `x` values to range from 0 to 25 (to correspond to the `pch` values). The `y` values are set at 1 so that you get a horizontal line of points; the `rep()` command is used to repeat the value 1 for 26 times. In other words, you get 26 1s to correspond to your various `x` values. You now set the plotting character to vary from 0 to 25 using `pch = 0:25`. Finally, you make the points a bit bigger using a character expansion factor (`cex = 2`). The `text()` command is used to add text to a current plot. You give the `x` and `y` coordinates of the text and the actual text you want to produce. In this instance, the `x` values were set to vary from 0 to 25 (corresponding to the plotted symbols). The `y` value was set to be 0.95 because this positions the text just under each symbol. Finally, you state the text you require; you want to produce a number here so you state the numbers you want (0 to 25) and make sure they are forced to be text using the `as.character`instruction.

The values 26 to 31 are not used, but values from 32 upward are; these are ASCIIcode.Thevalue32producesaspace,soitisnotveryusefulasaplotting character, but other values are fine up to about 127. You can also specify a character from the keyboard directly by enclosing it in quotes; to produce + symbols, for example, you type the following:

```
> plot(fw$speed, fw$count, pch = "+")
```

The + symbol is also obtained via `pch = 3`. You can alter the size of the plotted characters using the `cex =` instruction; this is a character expansion factor. So setting `cex = 2` makes points twice as large as normal and `cex = 0.5` makes them half normal size.

If you want to alter the color of the points, use the `col =` instruction and put the color as a name in quotes. You can see the colors available using the `colors()` command (it is quite a long list).

Setting Axis Limits

The `plot()` command works out the best size and scale of each axis to fit the plotting area. You can set the limits of each axis quite easily using `xlim =` and `ylim =` instructions. The basic form of these instructions requires two values—a start and an end:

```
xlim = c(start,
end) ylim =
c(start, end)
```

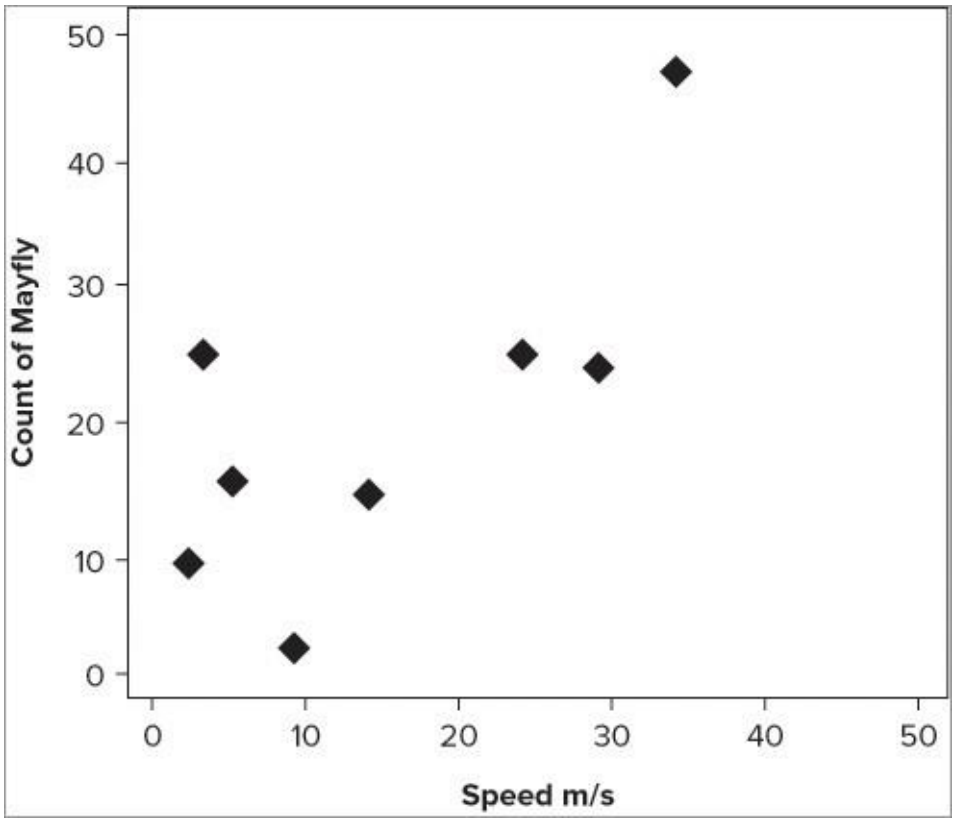
You can use these to force a plot to be square, for example, or perhaps to "zoom in" to a particular part of a plot or to emphasize oneaxis.

You can add all of these elements together to produce a plot that matches your particular requirements. In the current example, you might type the following `plot()` command:

```
> plot(fw$speed, fw$count, xlab = 'Speed m/s', ylab = 'Count
of Mayfly',
pch = 18, cex = 2, col = 'gray50', xlim = c(0, 50), ylim =
c(0, 50))
```

This is quite long, but you can break it down into its component parts. You always start with the `x` and then `y` values, but the other instructions can be in any order because they are named explicitly. The resulting scatter plot looks like Figure 7-11.

Figure 7-11:



Using Formula Syntax

There is another way that you can specify what you want to plot; rather than giving the `x` and `y` values as separate components, you produce a formula to describe thesituation:

```
> plot(count ~ speed, data = fw)
```

You use the tilde character (~) to symbolize your formula. On the left you place the response variable (that is, the dependent variable) and on the right you place the predictor (independent) variable. At the end you tell the command where to find these data. This is useful because it means you do not need to use the `$` syntax or use the `attach()` command to allow R to read the variables inside the data frame. This is the same formula you saw previously when looking at simple hypothesis tests in Chapter6.

Adding Lines of Best-Fit to ScatterPlots

In Chapter 5 you used the `abline()` command to add a straight line matching the slope and the intercept of a series of points when you produced a QQ plot. You can do the same thing here; first you need to determine the slope and intercept. You will look at the `lm()` command in more detail later (Chapter 10), but for now all you need to know is that it will work out the slope and intercept for you and pass it on to the `abline()` command.

```
> abline(lm(count ~ speed, data = fw))
```

Now you can see another advantage of using the formula notation: the command is very similar to the original `plot()` command. The default line produced is a thin solid black line, but you can alter its appearance in various ways. You can alter the color using the `col =` instruction, you can alter theline widthusingthe`lwd=`instruction;andyoucanalterthelinetypeusingthe`lty=` instruction.

The `lwd` instruction requires a simple numeric value—the bigger thenumber, the fatter the line! The `col` instruction is the same as you have met before and requires a color name in quotes. The `lty` instruction allows you to specify the line type in two ways: you can give a numeric value or a name in quotes. Table 7-1showsthevariousoptions.

Table 7-1: Line Types that Can be Specified Using the lty Instruction in a Graphical Command

Valu	Label	Result
0	blank	Blank
1	solid	Solid
2	dashed	Dashed
3	dotted	Dotted
4	dotdash	Dot-Dash
5	longdas	Long dash
6	twodas	Two dash

Notice that you can draw a blank line! The number values here are recycled, so if you use `lty = 7` you get a solid line (and then again with 13).

You can use these commands to customize your fitted line like so:

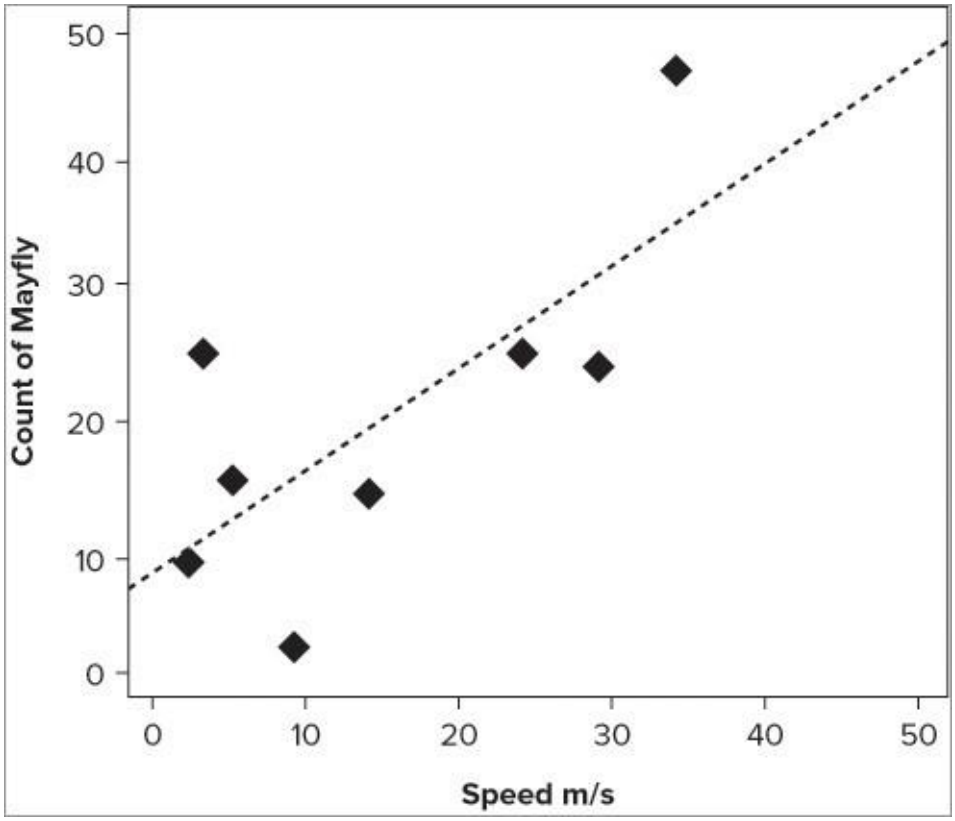
```
> abline(lm(count ~ speed, data = fw), lty = 'dotted', lwd =
2, col = 'gray50')
```

If you combine the previous `plot()` command with the preceding `abline()` command like so, you get something like Figure 7-12:

```
> plot(count ~ speed, data = fw, xlab = 'Speed
m/s', ylab = 'Count of Mayfly', pch = 18, cex =
2,
col = 'gray50', xlim = c(0, 50), ylim = c(0, 50))
> abline(lm(count ~ speed, data = fw), lty = 'dotted', lwd =
2, col = 'gray50')
```

The `plot()` command is very general and can be used by programmers to create customized graphs. You will mostly use it to create scatter plots, and in that regard it is still a flexible and powerful command. You can add additional graphical parameters to the `plot()` command. You see more of these additional instructions in Chapter 11, but for the time being the following activity gives you the opportunity to try making some scatter plots for yourself.

Figure 7-12:



The scatter plot is a useful tool for presentation of results and also in exploring your data. It can be useful, for example, to see a range of scatter plots for a data set all in one go as part of your data exploration. This is the focus of the next section.

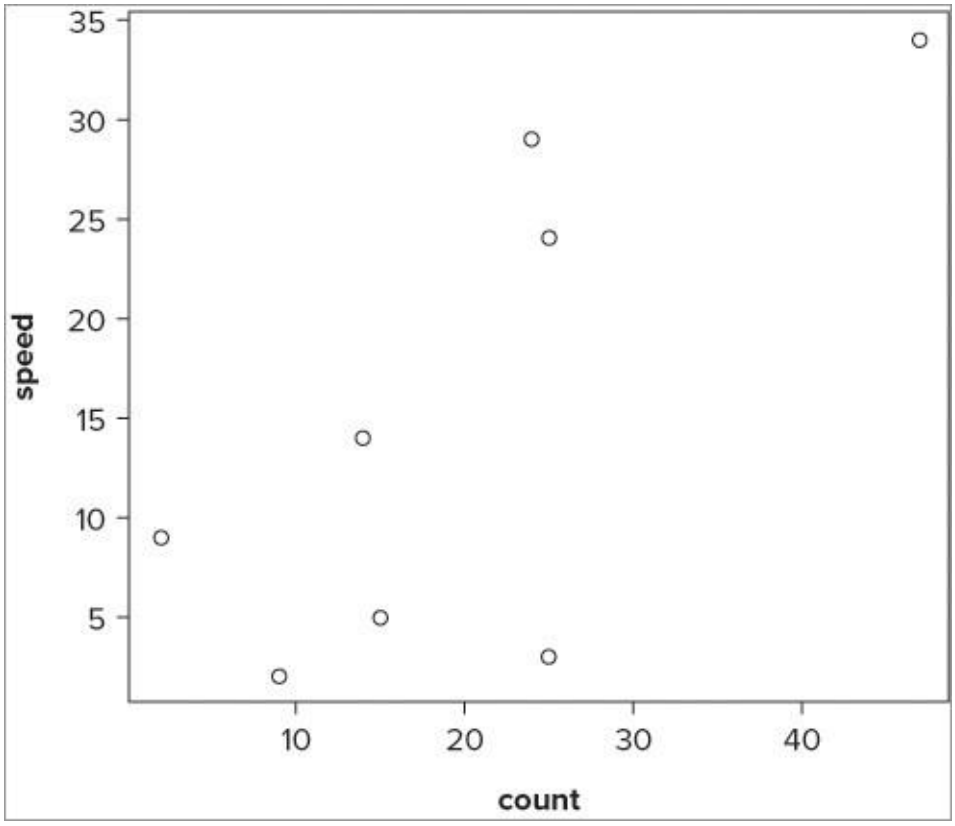
Pairs Plots (Multiple Correlation Plots)

In the previous section you looked at the `plot()` command as a way to produce a scatter plot. If you use the same data as before—two columns of numerical data —but do not specify the columns explicitly, you still get a plot of sorts:

```
> 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
> plot(fw)
```

This produces a graph like Figure 7-13.

Figure 7-13:



The data have been plotted, but if you look carefully you see that the axes are in a different order than the one you used before. The command has taken the first column as the `x` values, and the second column as the `y` values. If you try this on a data frame with more than two columns (as follows), you get something new, shown in the Figure7-14:

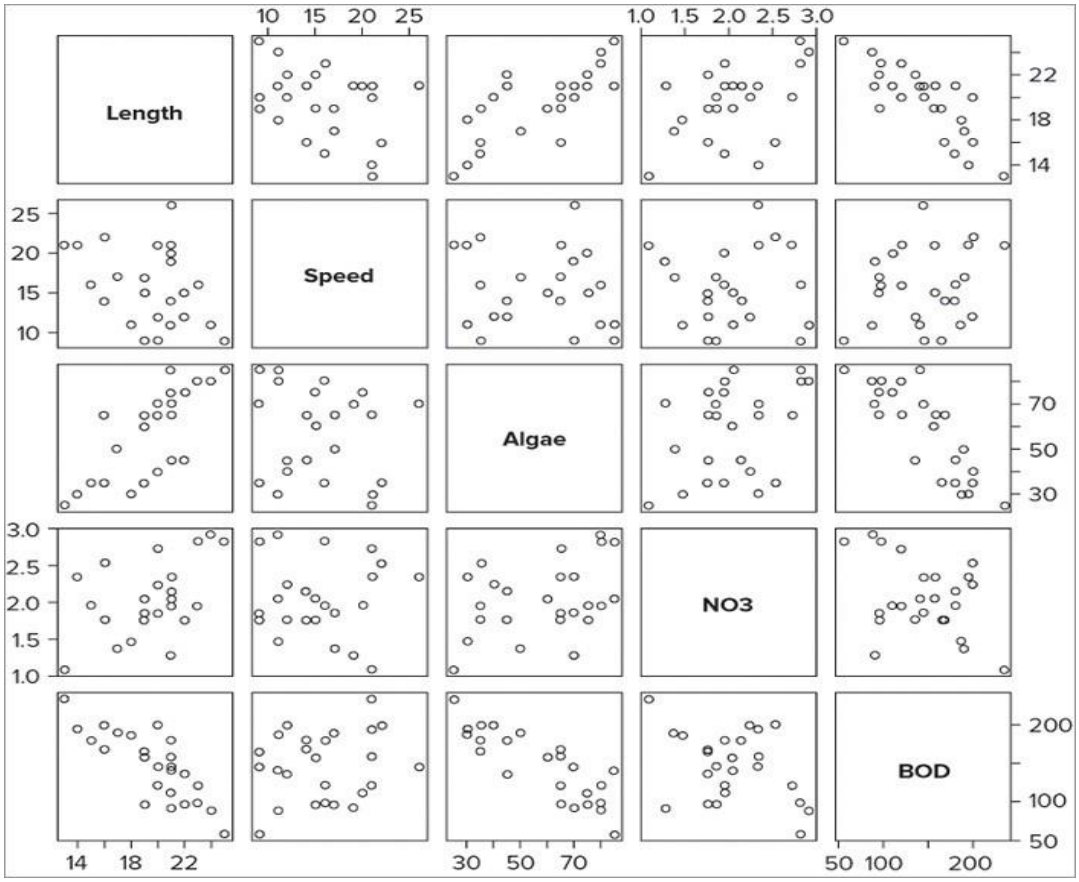
```
> head(mf)
  Length Speed Algae  NO3 BOD
1     20    12    40 2.25 200
2     21    14    45 2.15 180
3     22    12    45 1.75 135
4     23    16    80 1.95 120
5     21    20    75 1.95 110
6     20    21    65 2.75 120
> plot(mf)
```

You end up with a scatterplot matrix where each pairwise combination is plotted (refer to Figure 7-14). This has created a **pairs plot**—you can use a special command `pairs()` to create customized pairs plots.

By default, the `pairs()` command takes all the columns in a data frame and creates a matrix of scatter plots. This is useful but messy if you have a lot of columns. You can choose which columns you want to display by using the formula notation along the following lines:

```
pairs(~ x + y + z, data = our.data)
```

Figure 7-14:

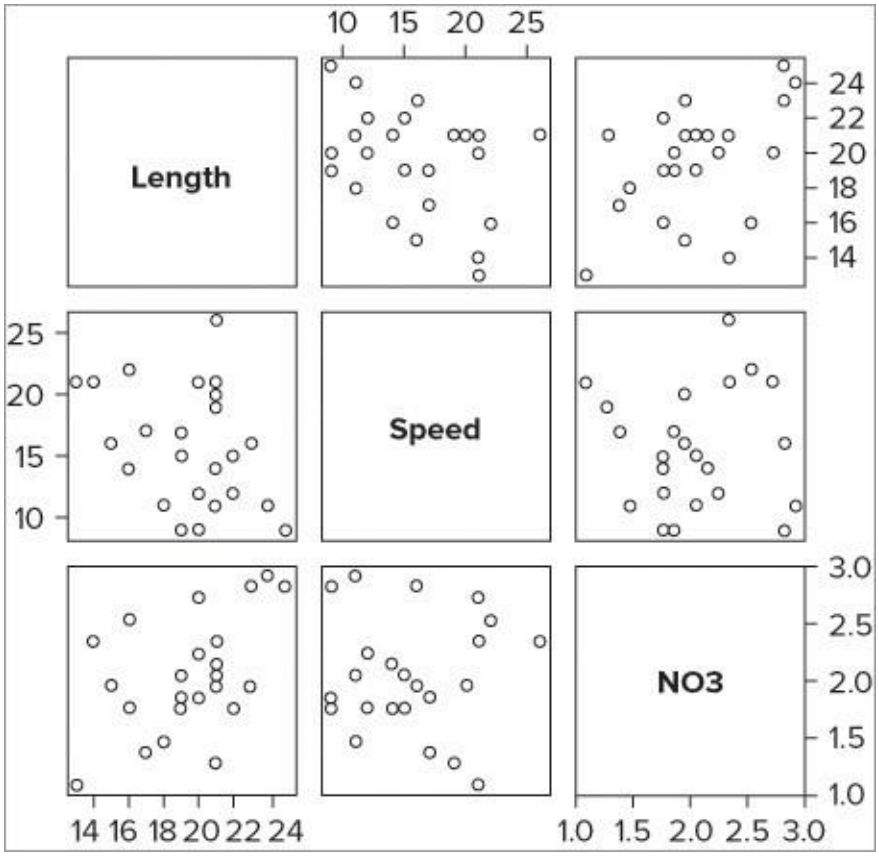


Your formula does not need anything on the left of the ~ because a response variable is somewhat meaningless in this context (this is like the `cor()` command you used in Chapter 6). You simply provide the required variables and separate them with + signs. If you are using a data frame, you also give the name of the data frame. In the current example you can select some of the columns likeso:

```
> pairs(~ Length + Speed + NO3, data = mf)
```

This produces a graph like Figure 7-15.

Figure 7-15:



You can alter the plotting characters, their size, and color using the `pch`, `cex`, and`col`instructionslikeyousawpreviously.Thefollowingcommandproduces largeredcrossesbutotherwiseisessentiallythesamegraph:

```
> pairs(~ Length + Speed + NO3, data = mf, col ='red', cex =
2, pch = 'X')
```

Itispossibletospecifyotherparameters,butitisfiendishlydifficultwithouta great deal of experience; the only parameters you can alter easily are the size of thelabelsonthediagonal,andthefontstyle.Toaltereitheroftheseyoucanuse thefollowinginstructions:

```
cex.labels = 2
font.labels = 1
```

The default magnification for the diagonal labels is 2; you can alter this by specifying a new value. You can also alter the font style: 1 = normal, 2 = bold, 3 = italic, and 4 = bold and italic.

Pairs plots are particularly useful for exploring your data. You can see several graphs in one window and can spot patterns that you would like to explore further. In the next section you look at another use for the `plot()` command: line charts.

Line Charts

So far you have looked at the `plot()` command as a way to produce scatter plots, either as a single pair of variables or a multiple-pairs plot. There may be many occasions when you have

data that is time-dependent, that is, data that is collected over a period of time. You would want to display these data as a scatter plot where the y-axis reflects the magnitude of the data you recorded and the x- axis reflects the time. It would seem sensible to be able to join the data together with lines in order to highlight the changes over time.

Line Charts Using NumericData

If the time variable you recorded is in the form of a numeric variable, you can use a regular `plot()` command. You can specify different ways to present the data using the `type` instruction. Table 7-2 lists the main options you can set using the `type`instruction.

Table7-2: Thetype=InstructionCanAltertheWayDataisDrawnonthePlot Area

Instructio	Explanation
`type =`	Points only.
`type =`	Points with line segments between.
`type =`	Lines segments alone with no points.
`type =`	Lines overplotted with points, that is, no gap between the line segments.
`type =`	Line segments only with small gaps where the points would be.
`type = 'n'`	Nothingisplotted!Thegraphisproduced,settingaxisscalesbutthedataarenotactuall y drawnin.

Therefore, if you want to highlight the pattern, you can specify `type = "l"` and draw a line, leaving the points out entirely. Notice that you can use `type = "n"` to produce nothing at all! This can be useful because it enables you to define the limits of a plot window, which you can add tolater.

Look at the `Nile` data that comes with R (simply type its name: `Nile`). This is stored as a special kind of object called a *time series*. Essentially, this enables you to specify the time in a more space-efficient manner than using a separate column of data. In the `Nile` data you have measurements of the flow of the Nile river from 1871 to 1970. If you plot these data you see something that resembles Figure7-16.

```
> plot(Nile, type = 'l')
```

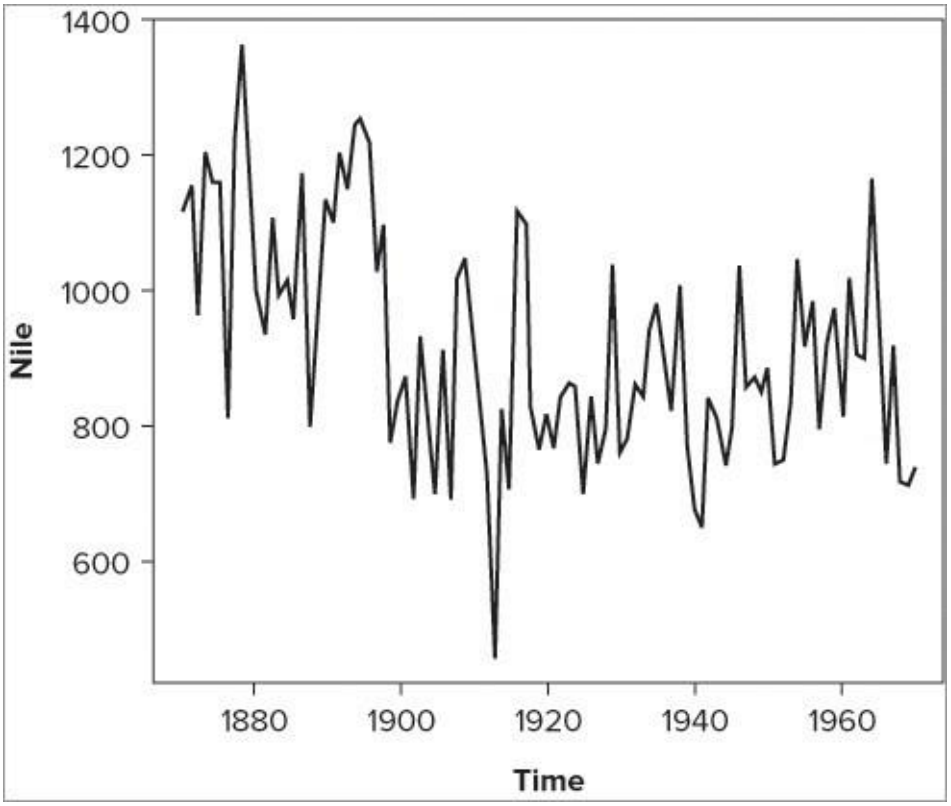
Here you specified `type = "l"`, which is the default for time series objects but not for regularobjects.

If your data are not in numerical order, you can end up with some odd-looking line charts. You can use the `sort()` command (recall Chapter 3) to reorder the data using the `x`-axis data, which usually *sorts* out the problem (pardon the pun). Look at the following examples:

```
> with(mf, plot(Length, NO3, type = 'l'))
> with(mf[order(mf$Length),], plot(sort(Length), NO3, type =
'l'))
```

In the first case the data are not sorted, and the result is a bit of a mess (the result is not shown here, but you can try it for yourself). In the second case the data are sorted, and the result is a lotbetter.

Figure 7-16:



Line Charts Using Categorical Data

If the data you have is a sequence but doesn't have a numerical value, you have a trickier situation. For example, your time interval might be recorded as month of the year. In this case you can think of a line plot as a special case of a scatter

plot, but where one axis (the dependent/x-axis) is not a numeric scale but a categorical one. The following data provides an example; in this case you have numeric data with labels that are categorical (each being a month of the year):

```
> rain
Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov
  Dec 3   5   7   5   3   2   6   8   5   6
      9   8
```

Alternatively, you might have data in the form of a data frame; the following example shows the same data but this time the labels are in a second column:

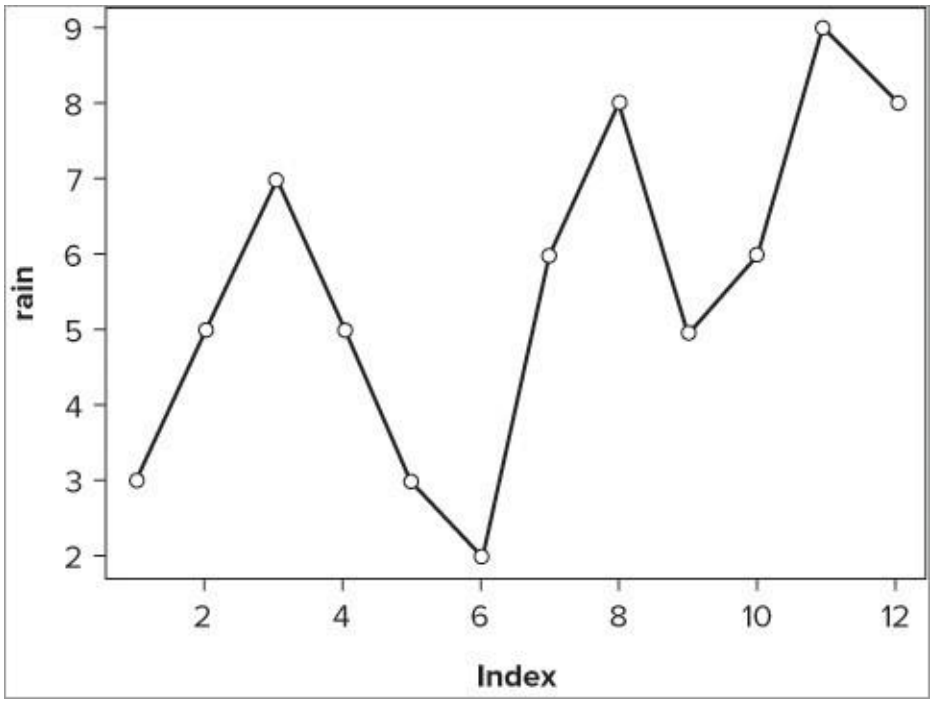
```
> rainfall
   rain month
1     3   Jan
2     5   Feb
3     7   Mar
4     5   Apr
5     3   May
6     2   Jun
7     6   Jul
8     8   Aug
9     5   Sep
10    6   Oct
11    9   Nov
12    8   Dec
```

In either case, you could try plotting the data using the `plot()` command like so:

```
plot(rain, type = 'b')
plot(rainfall$rain, type =
'b')
```

In the first instance of the command you simply type the name of the data vector; in the second case you have to use the `$` syntax to get the data from within the data frame. As part of the `plot()` command you add the instruction `type = 'b'`; this creates both points and lines. You do get a plot of sorts (see Figure 7-17), but you do not have the months displayed; the x-axis remains asa simple numericindex.

Figure 7-17:



To alter the x-axis as desired you need to remove the existing x-axis, and create your own using the character vector as the labels. Perform the following steps to do so:

1. Startbyturningofftheaxesusingthe`axes=FALSE`instruction.Youcan still label the axes using the `xlab` and `ylab` instructions as you have seen before.Ifyouwanttoproduceblanklabelsandaddthemlaterusingthe `title()` command, set them using a pair of quotes; for example, `xlab =""`:

```
> plot(rain, type = 'b', axes = FALSE, xlab = 'Month', ylab =
'Rainfall  cm')
```

This makes a line plot with appropriately labeled axes, but no actual axes!

2. Now construct your x-axis using the character labels you already have (or you could make new labels). The `axis()` command creates an axis for a plot. The basic layout of the command is like so:

```
axis(side, at = NULL, labels
=TRUE)
```

The first part is where you set which side you want the axis to be created on; 1 is the bottom, 2 is the left, 3 is the top, and 4 is the right side of the plot. The `at =` part is where you determine how many tick marks are to be shown; you show this as a range from 1: *n* where *n* = how many tick marks you require, (12 in thiscase).

3. Finally, you get to point to the labels. In this example you use a separate character vector for thelabels:

```
> month = c('Jan', 'Feb', 'Mar', 'Apr', 'May',
'Jun', 'Jul', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec')
> axis(side = 1, at = 1: length(rain), labels
=month)
```

This creates an axis at the bottom of the plot (the x-axis) and sets the tick marks from 1 to 12; you use the `length()` command to make sure you get the correct number, but you could have typed `1:12` instead. Finally, you point to the month vector to get the labels; if these were contained as row names or column names you could use the appropriate command to get them: `row.names()` or `names()`, for example. If they were in a separate column, you point to them using the `$` syntax: for example, `rainfall$month`.

4. Tofinishoffyourplot,makethey-axis.Youcanmakethey-axisusing:

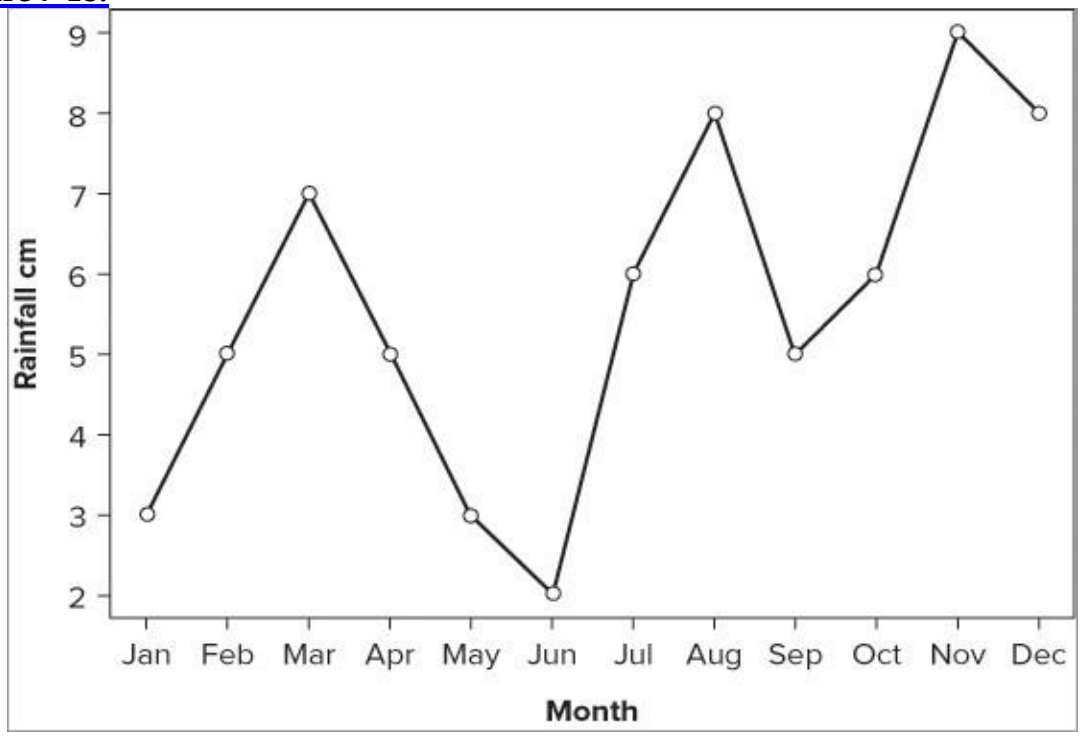
```
> axis(side = 2)
```

This creates an axis for you and takes the scale from the existing plot.

```
> plot(rain, type = 'b', axes = FALSE, xlab = 'Month', ylab =
'Rainfall cm')
> month = c('Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun',
'Jul', 'Aug', 'Sep',
'Oct', 'Nov', 'Dec')
> axis(side = 1, at = 1: length(rain), labels = month)
> axis(side = 2)
> box()
```

The final plot looks like Figure 7-18.

Figure 7-18:



You can alter the plotting characters and the characteristics of the line using instructions that you have seen before: `pch` alters the plotting symbol, `cex` alters the symbol size, `lty` sets the line type, and `lwd` makes the line wider or thinner. You can use the `col =` instruction to specify a color for the line and points (both set via the sameinstruction).

Pie Charts

If you have data that represents how something is divided up between various categories, the pie chart is a common graphic choice to illustrate your data. For example, you might have data that shows sales for various items for a whole year. The pie chart enables you to show how each item contributed to total sales. Each item is represented by a slice of pie—the bigger the slice, the bigger the contribution to the total sales. In simple terms, the pie chart takes a series of data, determines the proportion of each item toward the total, and then represents these as differen slices of thepie.

```
> data11
 [1] 3 5 7 5 3 2 6 8 5 6 9 8
```

When you use the `pie()` command, these values are converted to proportions of the total and then the angle of the pie slices is determined. If possible, the slices are labeled with the names of the data. In the current example you have a simple vector of values with no names, so you must supply them separately. You can do this in a variety of ways; in this instance you have a vector of character labels:

```
> data8
 [1] "Jan" "Feb" "Mar" "Apr" "May" "Jun" "Jul" "Aug" "Sep"
"Oct" "Nov" "Dec"
```

To create a pie chart with labels you use the `pie()` command in the following manner:

```
> pie(data11, labels = data8)
```

This produces a plot that looks like Figure 7-19.

You can alter the direction and starting point of the slices using the `clockwise =` and `init.angle =` instructions. By default the slices are drawn counter- clockwise, so `clockwise = FALSE`; you can set this to `TRUE` to produce clockwise slices. The starting angle is set to 0º (this is 3 o'clock) by default when you have `clockwise = FALSE`. The starting angle is set to 90º (12 o'clock) when you have `clockwise = TRUE`. To start the slices from a different point, you simply give the starting angle in degrees; these may also be negative with `-90` being equivalent to270º.

The default colors used are a range of six pastel colors; these are recycled as necessary. You can specify a range of colors to use with the `col =` instruction. One way to do this is to make a list of color names. In the following example you make a list of gray colors and then use these for your chartedcolors:

```
> pc = c('gray40', 'gray50', 'gray60', 'gray70',
'gray80', 'gray90')
> pie(data11, labels = data8, col = pc, clockwise =
TRUE, init.angle = 180)
```

You can also set the slices to be drawn clockwise and set the starting point to 180º, which is 9 o'clock. The resulting plot looks like Figure 7-20.

Figure 7-19:

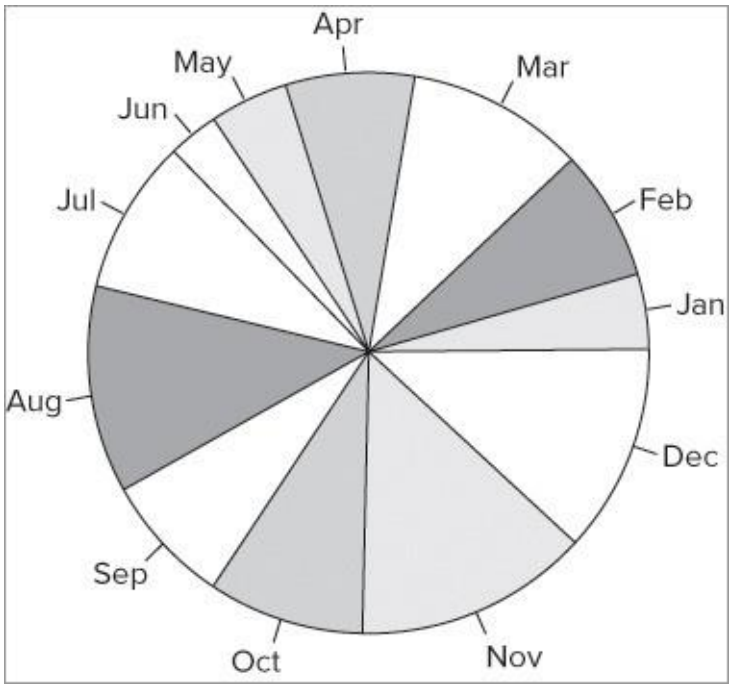
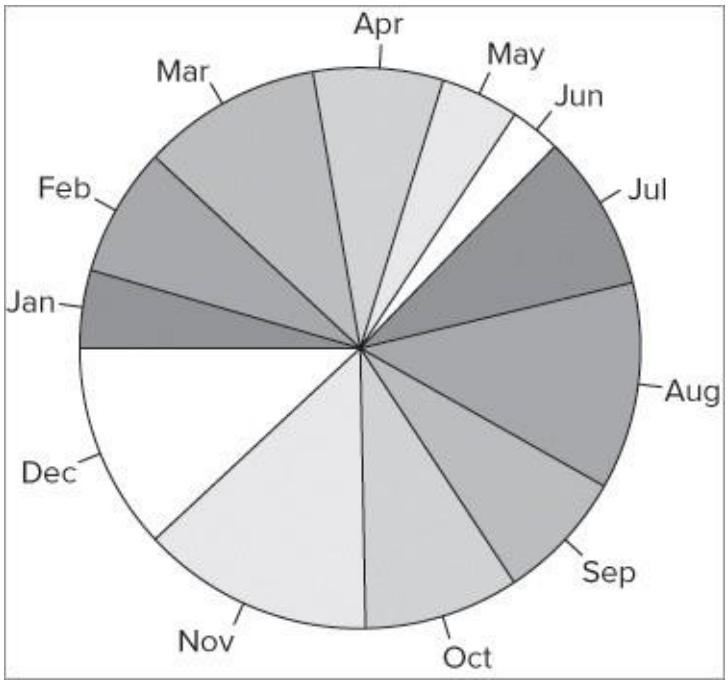


Figure 7-20:



When your data are part of a data frame, you must use the `$` syntax to access the column you require or use the `with()` or `attach()` commands. In the following example, the data frame contains row names you can use to label your pieslices:

```
> 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
```

```
> pc = c('gray65', 'gray70', 'gray75', 'gray80',
'gray85', 'gray90')
> pie(fw$count, labels = row.names(fw), col = pc, cex = 1.2)
```

In this case you set the colors to six shades of gray and also use the `cex =` instruction to make the slice labels a little bigger. The `labels =` instruction points to the row names of the data frame. The final graph looks like Figure 7-21.

shows a matrix of bird observation data; the rows and the columns are named:

```
> 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            46       16        8       4        0
Robin             9        3        0       0        2
Song Thrush       4        0        6       0        0
```

You can use the `[row, column]` syntax with the `pie()` command; here you examine the firstrow:

```
> pie(bird[,1], col = pc)
```

This produces a graph like Figure 7-22; note that you use the same gray colors that you created earlier as your color palette:

Figure 7-21:

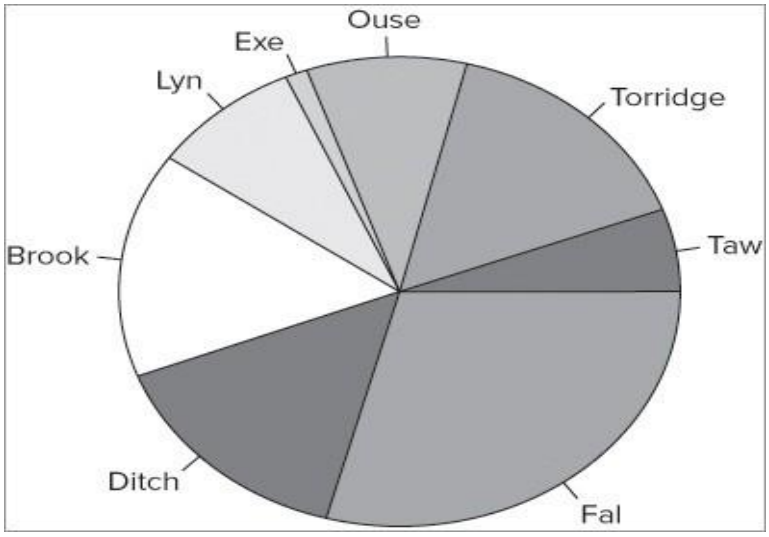
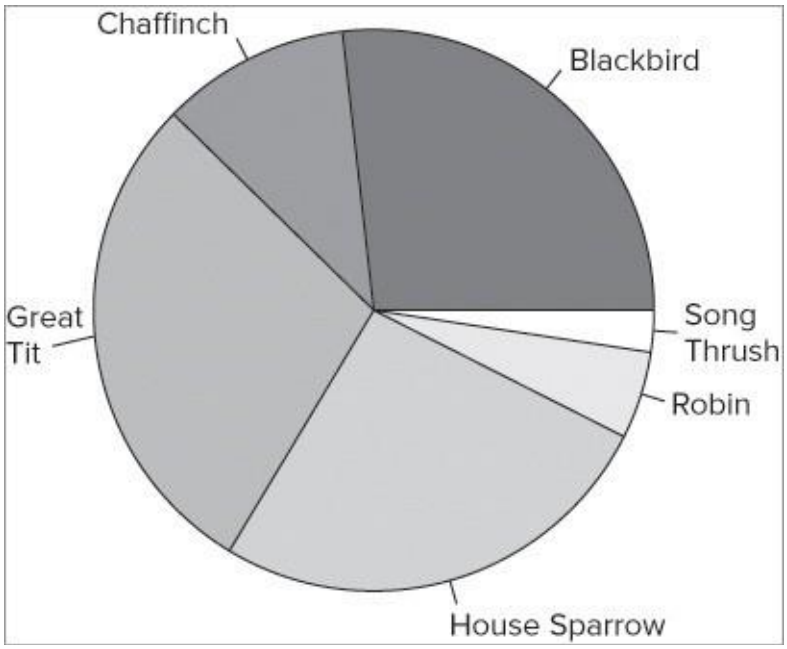


Figure 7-22:



If you have your data in a data frame rather than a matrix, you get an error message like the following when you try to pie chart a row:

```
> mf[1,]
  LengthSpeedAlgae    NO3BOD
1     20    12    40 2.25200

> pie(mf[1,])
Error in pie(mf[1, ]) : 'x' values must be positive.
```

When you look at the row in question, it looks as though it ought to be fine but it is not. You can make it work by converting the data into a matrix. You can do this transiently using the `as.matrix()` command. In the following example you see a data frame and the command used to make a pie chart from the firstrow:

```
> head(mf)
  Length Speed Algae  NO3 BOD
1     20    12    40 2.25 200
2     21    14    45 2.15 180
3     22    12    45 1.75 135
4     23    16    80 1.95 120
5     21    20    75 1.95 110
6     20    21    65 2.75 120


> pie(as.matrix(mf[1,]), labels = names(mf), col =
pc)
```

You can, of course, make pie charts from the columns, in which case you specify the column you require using the `[row, column]` syntax. The following command examples

both produce a pie chart of the `Hedgerow`column in the `bird`

data you saw previously:

```
> pie(bird[,2])
> pie(bird[,'Hedgerow'])
```

You can add other instructions to the `pie()` command that will give you more control over the final graph. You learn some of these later in Chapter 11. Next you look at the Cleveland dot plot.

Cleveland Dot Charts

An alternative to the pie chart is a Cleveland dot plot. All data that might be presented as a pie chart could also be presented as a bar chart or a dot plot. You can create Cleveland dot plots using the `dotchart()` command. If your data are a simple vector of values then like the `pie()` command, you simply give the vector name. To create labels you need to specify them. In the following example you have a vector of numeric values and a vector of character labels; you met these earlier when making a pie chart:

```
> data11; data8
 [1] 3 5 7 5 3 2 6 8 5 6 9 8
 [1] "Jan" "Feb" "Mar" "Apr" "May" "Jun" "Jul" "Aug" "Sep"
"Oct" "Nov" "Dec"

> dotchart(data11, labels = data8)
```

The resulting dot plot looks like Figure 7-23.

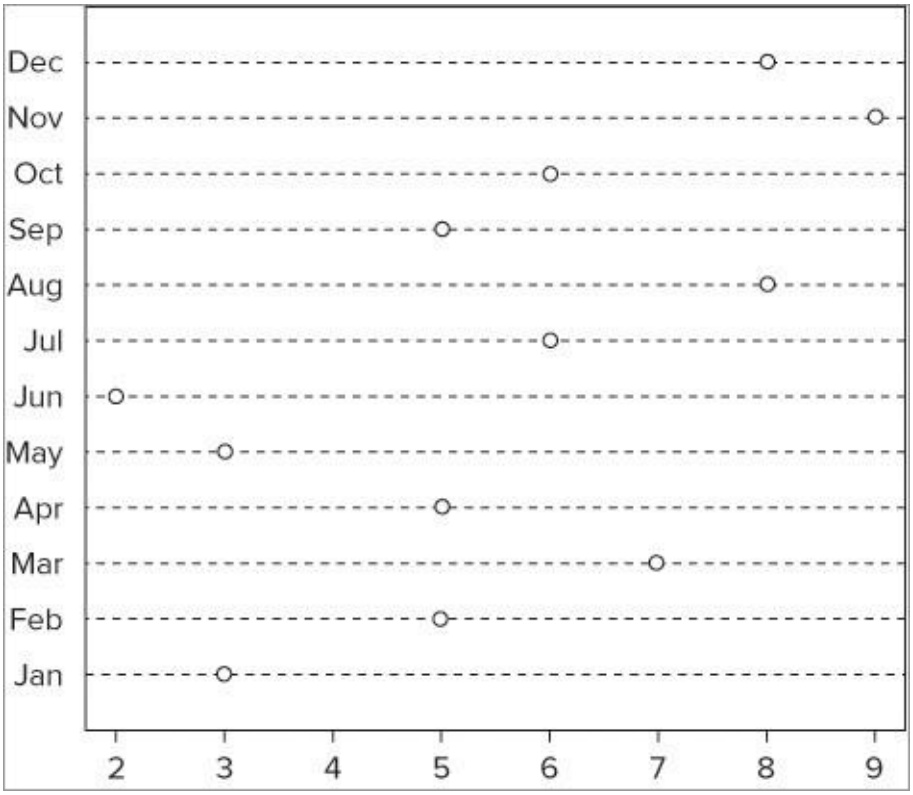
You can alter various parameters, but first you look at a more complex data example. Your data are best used if they are in the form of a matrix; the following data are bird observations that you used in previous examples:

```
> 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            46       16        8       4        0
Robin             9        3        0       0        2
Song Thrush       4        0        6       0        0
```

With a pie chart you must create a pie for the rows or the columns separately; with the dot plot you can do both at once. You can create a basic dot plot grouped by columns simply by specifying the matrix name like so:

```
> dotchart(bird)
```

Figure 7-23:



This produces a dot plot that looks like Figure 7-24.

Here you see the data shown column by column; in other words, you see the data for each column broken down by rows. You might choose to view the data in a different order; by transposing the matrix you could display the rows as groups, broken down by column:

```
> dotchart(t(bird))
```

You use the `t()` command to transpose the matrix and produce your dot plot, which looks like Figure 7-25.

You can alter a variety of parameters on your plot. Table 7-3 illustrates a few of theoptions.

Table7-3: SomeoftheAdditionalGraphicalInstructionsforthedotchart() Command

Instruction	Explanation
`color = 'color na`	Specifies the color to use for the plotted points and the main labels.
`gcolor = 'color na`	Specifies the color to use for the group labels and group data points (if specified).
`gdata = group.data`	You can specify a value to show for each group. This will typically be a mean.

lcolor =	Specifies the color to use for the lines across the chart.
cex = 1	Sets the character expansion factor for points and all the labels on the axes.
xlab = 'text.label'	Youcanspecifyalabel/titleforthex-axis.Youcanalsospecifyoneforthey-axis,butthis will usually overlap the labels from the data. Use the title() command afterwards to placeanaxislabel/title.
xlim =	Sets the limits of the x-axis. You specify the start and end points.
bg = 'color.na	Setsthebackgroundcolorfortheplottingsymbols.Thisworksonlywithanopenstyle of symbol.
pch = 21	Setstheplottingcharacter.Useanumericalvalueoracharacterfromthekeyboardin quotes.

Figure 7-24:

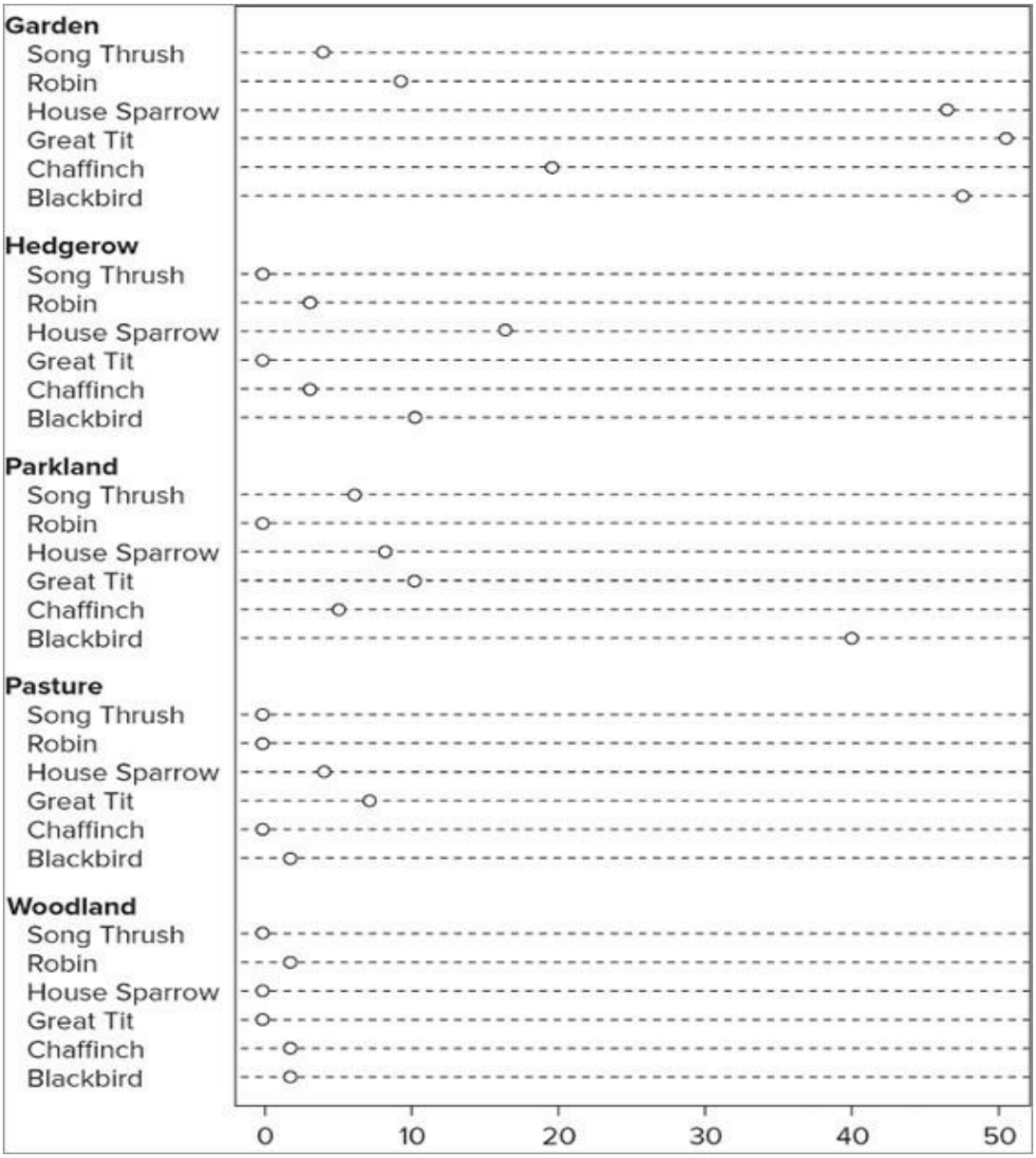
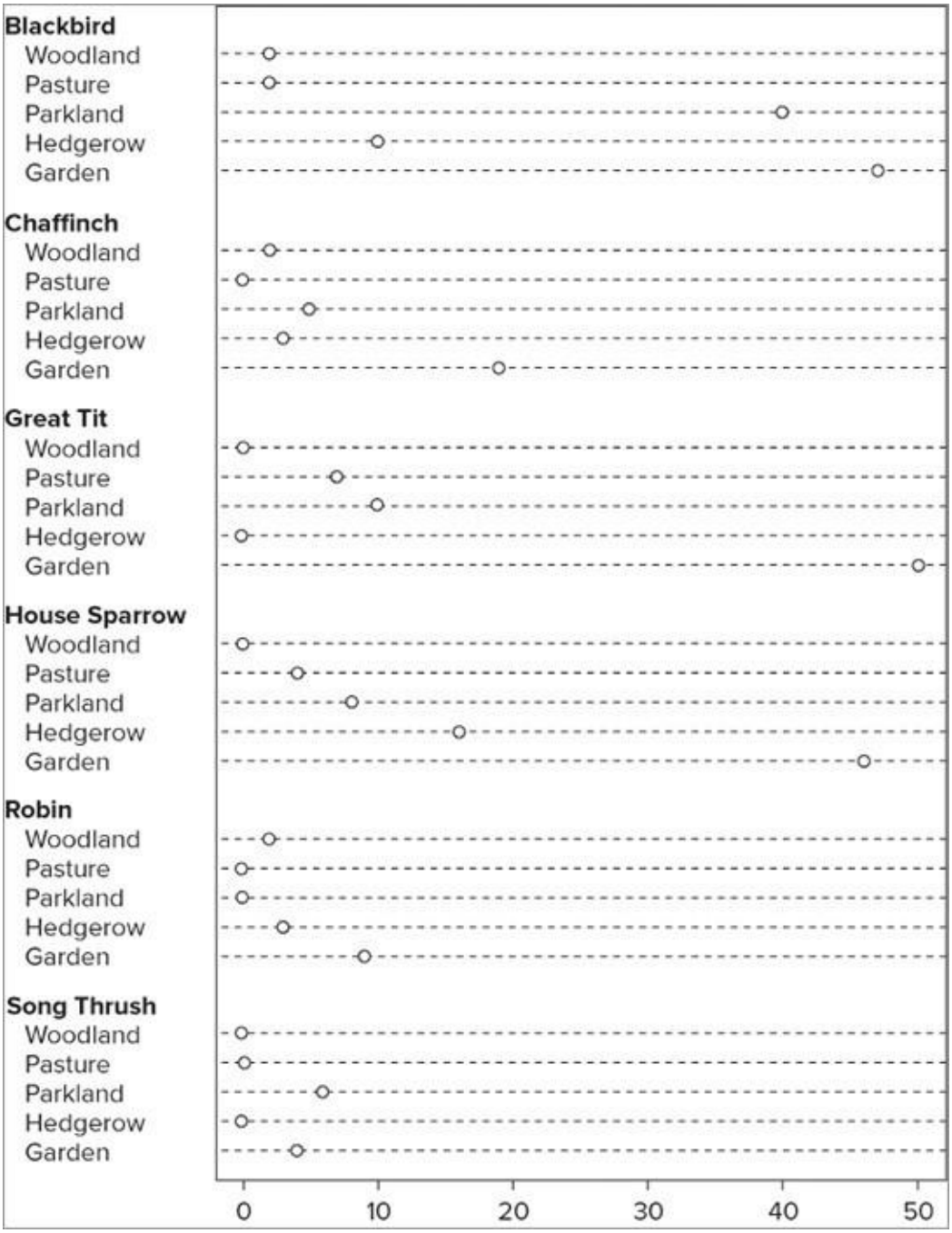

Garden
Song Thrush
Robin
House Sparrow
Great Tit
Chaffinch
Blackbird
Hedgerow
Song Thrush
Robin
House Sparrow
Great Tit
Chaffinch
Blackbird
Parkland
Song Thrush
Robin
House Sparrow
Great Tit
Chaffinch
Blackbird
Pasture
Song Thrush
Robin
House Sparrow
Great Tit
Chaffinch
Blackbird
Woodland
Song Thrush
Robin
House Sparrow
Great Tit
Chaffinch
Blackbird
0
10
20
30
40
50

Figure 7-25:



The following command utilizes some of these instructions to produce the graph shown in Figure 7-26:

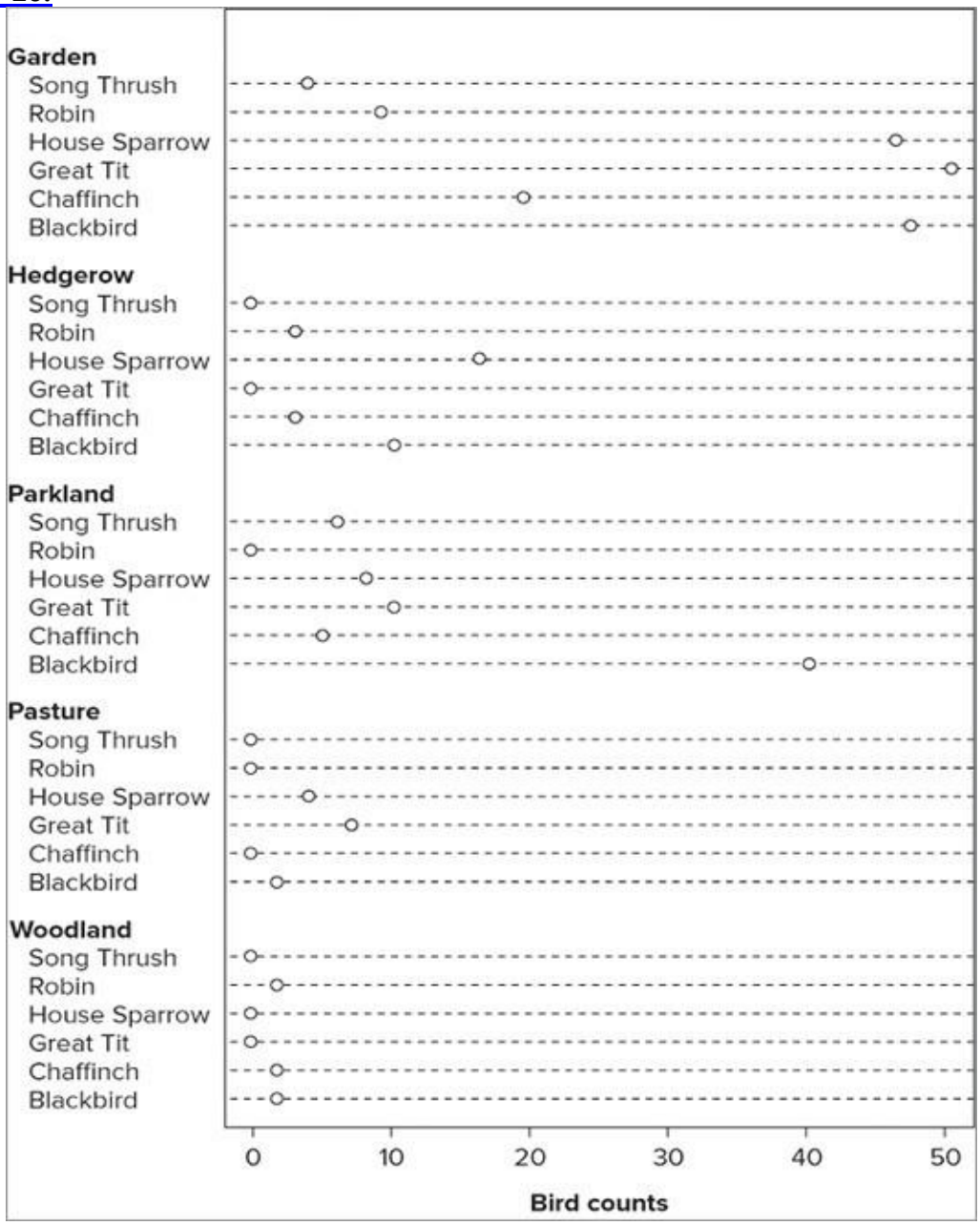
```
> dotchart(bird, color = 'gray30', gcolor = 'black',
lcolor = 'gray30',
cex = 0.8, xlab = 'Bird Counts', bg = 'gray90', pch = 21)
```

You can also specify a mathematical function to apply to each of the groups using the `gdata =` instruction. It makes the most sense to use an average of some kind—mean or median— to do so. In the following example the mean is used as a groupingfunction:

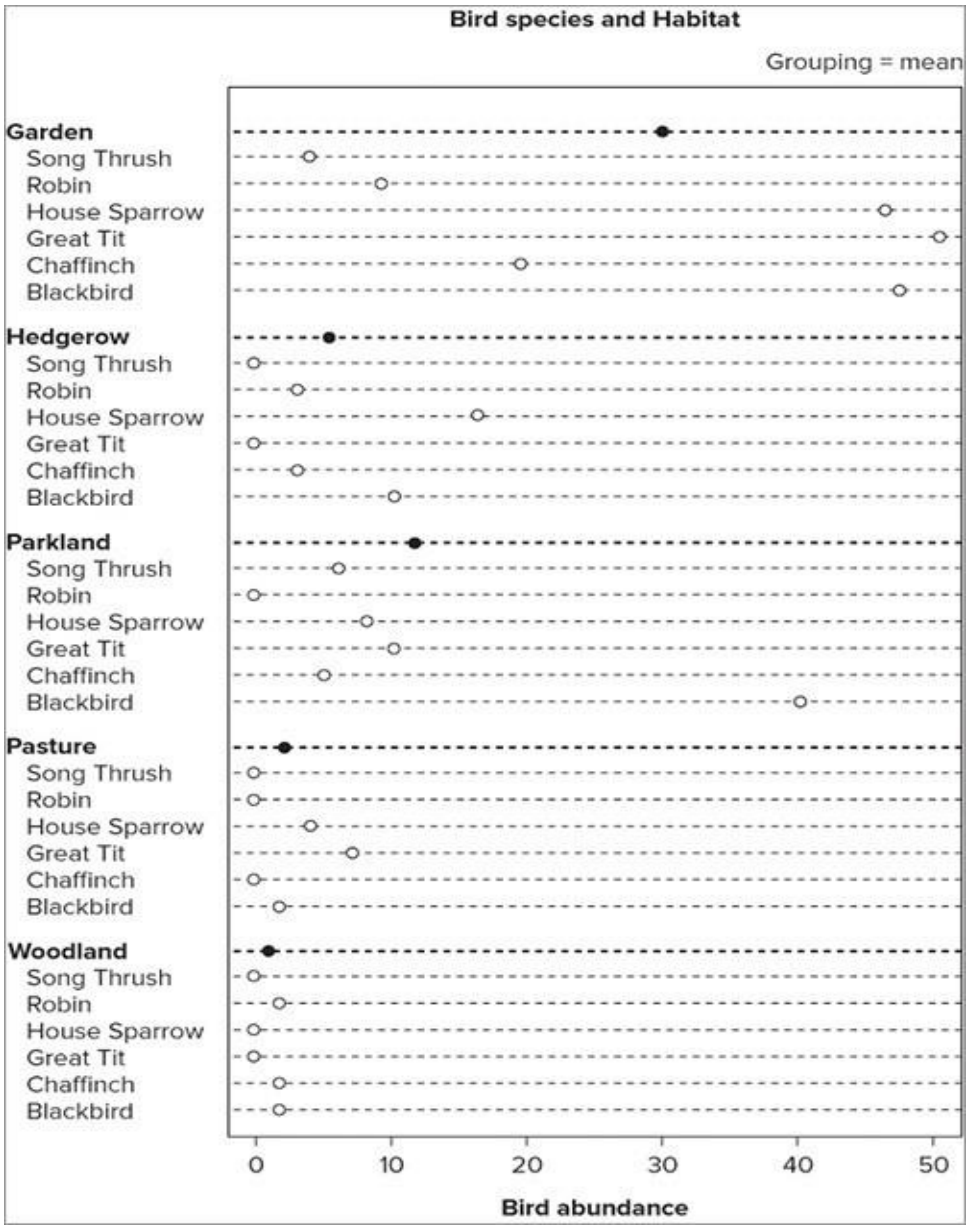
```
> dotchart(bird, gdata = colMeans(bird), gpch = 16,
```

```
gcolor = 'blue')
> mtext('Grouping = mean', side =3, adj = 1)
> title(main = 'Bird species and Habitat')
> title(xlab = 'Bird abundance')
```

Figure 7-26:



The first line of command draws the main plot; the mean is taken by using the `colMeans()`



command and applying it to the plot via the `gdata =` instruction. You can specify this function in any way that produces the values you require, and you can simply specify the values explicitly using the `c()` command.

The plotting character of the grouping function is set using the `gpch =` instruction; here, a filled circle is used to make it stand out from the main points. The `gcolor =` instruction sets a color for the grouping points (and labels).

The second line adds some text to the margin of the plot; here you use the top axis (`side = 1` is the bottom, `2` is the left) and adjust the text to be at the extreme end (`adj = 0` would be at the other end of the axis). The final two lines add titles to the main plot and the value axis (the x-axis); the resulting plot looks like Figure7-27.

The `mtext()` command is explored more thoroughly in Chapter 11, where you learn more about customizing and tweaking your graphs.

The Cleveland dot chart is a powerful and useful tool that is generally regarded as a better alternative to a pie chart. Data that can be presented as a pie chart can also be shown as a bar chart, and this type of graph is one of the most commonly usedgraphsformanypurposes.Barchartsarethesubjectofthenextsection.

Bar Charts

The bar chart is suitable for showing data that fall into discrete categories. In Chapter 3, "Starting Out: Working with Objects," you met the histogram, which is a form of bar chart. In that example each bar of the graph showed the number of items in a certain range of data values. Bar charts are widely used because they convey information in a readily understood fashion. They are also flexible and can show items in variousgroupings.

You use the `barplot()` command to produce bar charts. In this section you see how to create a range of bar charts, and also have a go at making some for yourself by following the activity at theend.

Single-Category Bar Charts

The simplest plot can be made from a single vector of numeric values. In the following example you have such an item:

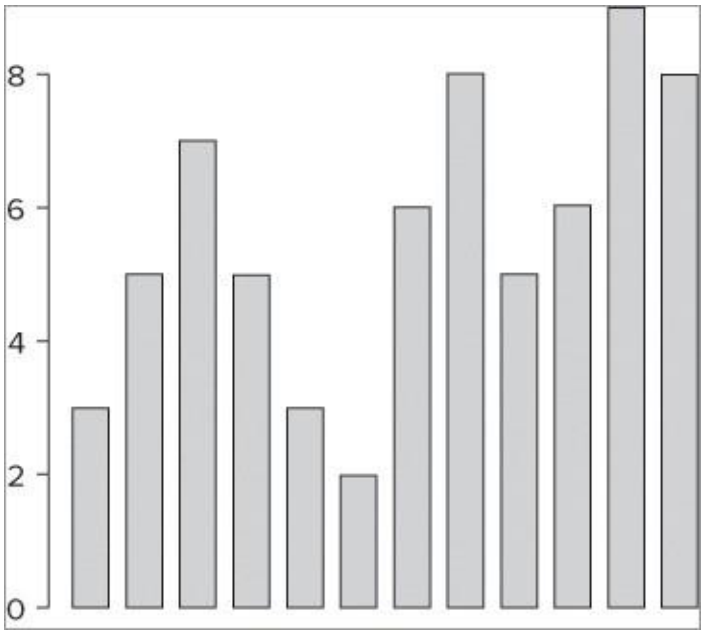
```
> rain
 [1] 3 5 7 5 3 2 6 8 5 6 9 8
```

To make a bar chart you use the `barplot()` command and specify the vector name in the instruction like so:

```
barplot(rain)
```

This makes a primitive plot that looks like Figure 7-28.

Figure 7-28:



Thecharthasnoaxislabelsofanykind,butyoucanaddthemquitesimply.To startwith,youcanmakenamesforthebars;youcanusethe`names=`instruction topointtoavectorofnames.Thefollowingexampleshowsonewaytodothis:

```
> rain
 [1] 3 5 7 5 3 2 6 8 5 6 9 8
> month
 [1] "Jan" "Feb" "Mar" "Apr" "May" "Jun" "Jul" "Aug" "Sep"
"Oct" "Nov" "Dec"
> barplot(rain, names = month)
```

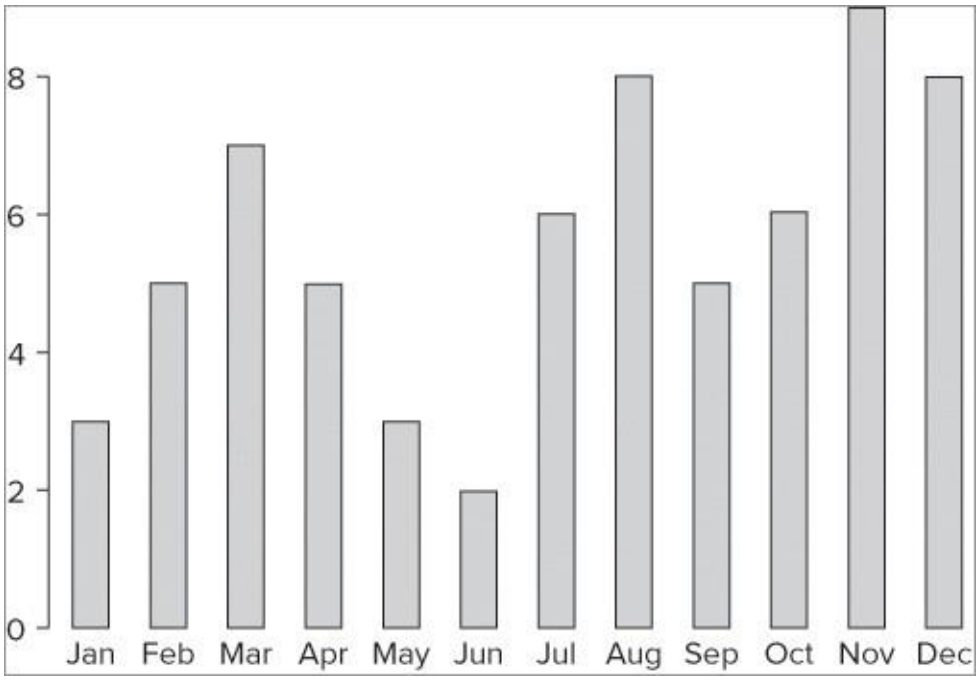
In this case you already had a vector of names; if you did not, you could make one or simply specify the names using a `c()` command like so:

```
> barplot(rain, names = c('Jan', 'Feb', 'Mar', 'Apr',
'May', 'Jun', 'Jul',
'Aug', 'Sep', 'Oct', 'Nov', 'Dec'))
```

If your vector has a `names` attribute, the `barplot()` command can read the names directly. In the following example you set the `names()` of the `rain` vector and then use the `barplot()` command:

```
> rain ; month
 [1] 3 5 7 5 3 2 6 8 5 6 9 8
 [1] "Jan" "Feb" "Mar" "Apr" "May" "Jun" "Jul" "Aug" "Sep"
```

Figure 7-29:



To add axis labels you can use the `xlab` and `ylab` instructions. You can use these as part of the command itself or add the titles later using the `title()` command. In the following example you create axis titles afterwards:

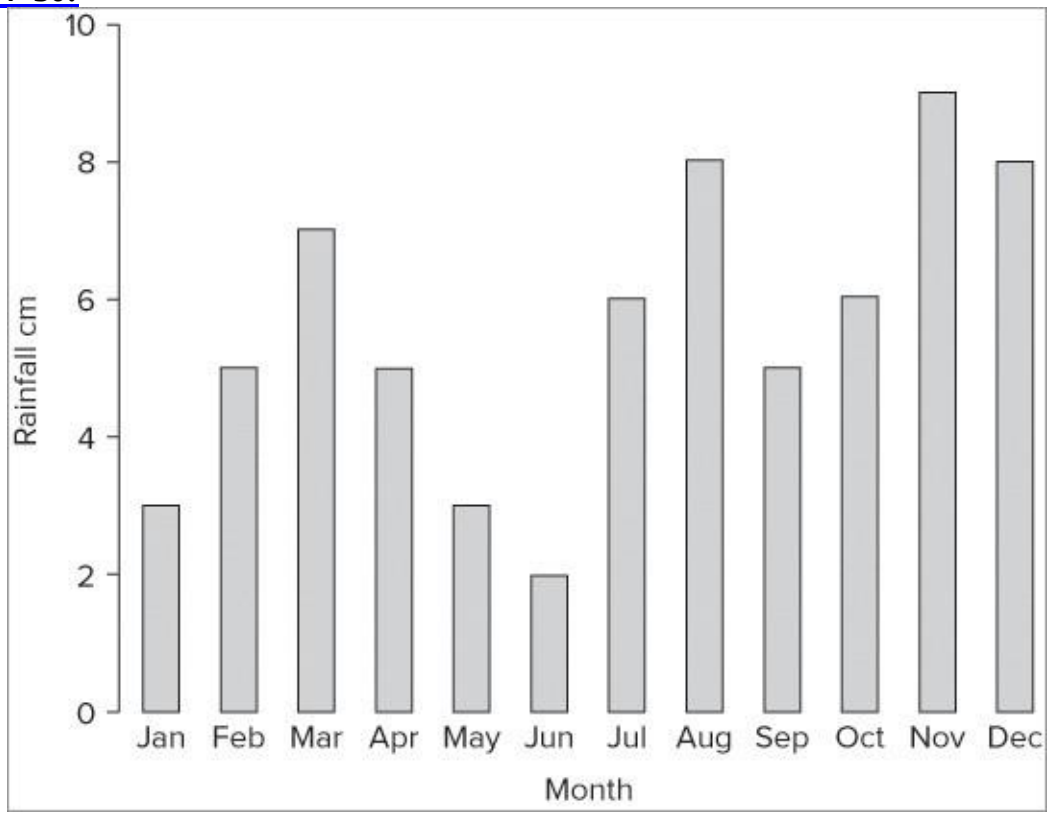
```
> barplot(rain)
> title(xlab = 'Month', ylab = 'Rainfall cm')
```

The y-axis is a bit short in Figure 7-29. You can alter the y-axis scale using the `ylim`instruction as shown in the following example:

```
> barplot(rain, xlab = 'Month', ylab = 'Rainfall cm',
ylim = c(0,10))
```

Recall that you need two parts to set the y-axis limit: a starting point and an ending point. Once you implement these two parts your plot looks more reasonable (see Figure7-30).

Figure 7-30:



You can alter the color of the bars using the `col =` instruction. If you want to "ground" the plot, you could add a line under the bars using the `abline()` command:

```
> abline(h = 0)
```

Inotherwords,youaddahorizontallineat0onthey-axis.Ifyouwouldrather have the whole plot enclosed in a box, you could use the `box()` command.You canalsousethe`abline()`commandtoaddgridlines:

```
> abline(h = seq(1, 9, 2), lty = 2, lwd = 0.5, col = 'gray70')
```

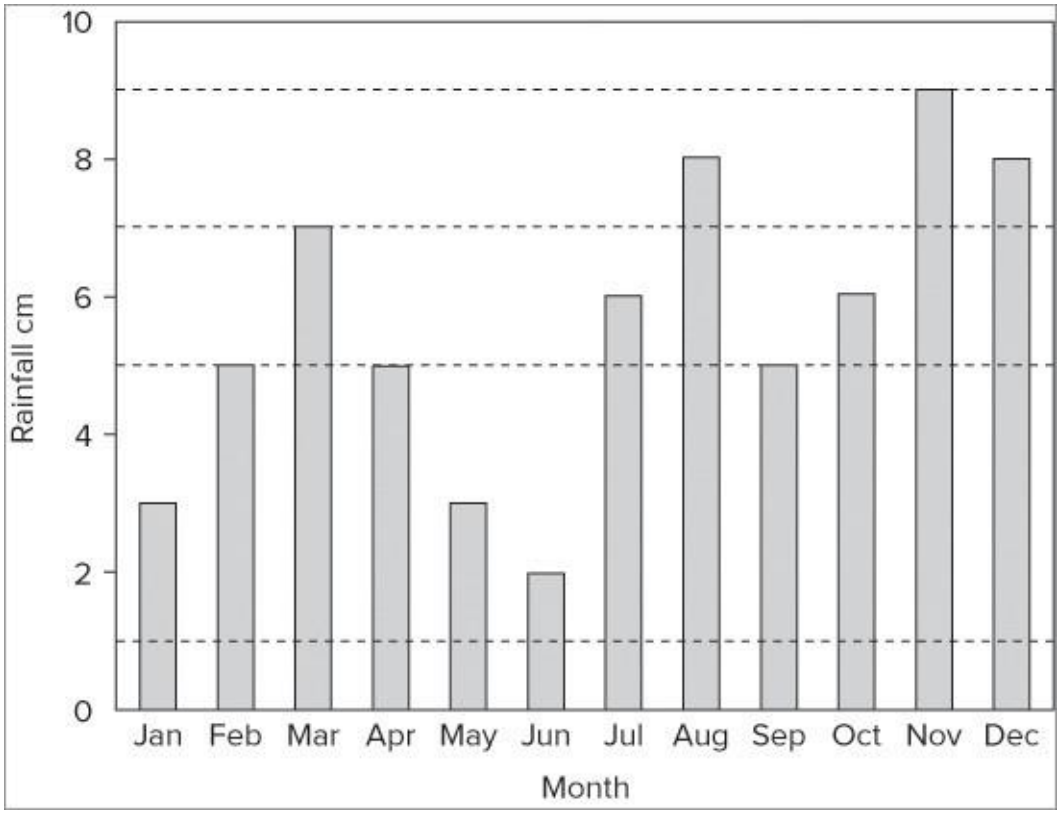
In this example you create horizontal lines using a sequence, the `seq()` command. With this command you specify the starting value, the ending value, and the interval. The `lty =` instruction sets the line to be dashed, and the `lwd =` instruction makes the lines a bit thinner than usual. Finally, you set the gridline colors to be a light gray using the `col =` instruction.

When you put the commands together, you end up with something like this:

```
> barplot(rain, xlab = 'Month', ylab = 'Rainfall cm',
ylim = c(0,10),
col = 'lightblue')
> abline(h = seq(1,9,2), lty = 2, lwd = 0.5, col = 'gray40')
> box()
```

The final graph looks like Figure 7-31.

Figure 7-31:



Previously you looked at the `hist()` command as a way to produce a histogram of a vector of numeric data. You can create a bar chart of frequencies that is superficially similar to a histogram by using the `table()` command:

```
>
table(rain)
rain
2 3 5 6 7 8 9
1 2 3 2 1 2 1
```

Here you see the result of using the `table()` command on your data; they are split into a simple frequency table. The first row shows the categories (each relating to an actual numeric value), and the second row shows the frequencies in each of these categories. If you create a `barplot()` using these data, you get something like Figure 7-32, which is produced using the following commands:

```
> barplot(table(rain), ylab = 'Frequency', xlab =
```

```
  'Numeric category')
  > abline(h = 0)
```

In Figure 7-32 you also added axis labels and drew a line under the bars with the `abline()` command.

When your data are part of a data frame, you must extract the vector you require using the `$` syntax or the `attach()` or `with()` commands. In the following example you see a data frame with two columns. In this case you also have row names, and you can use these to create name labels for the bars:

```
> 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
```

```
> barplot(fw$count, names = row.names(fw), ylab =
'Invertebrate Count' ,
col = 'tan')
> abline(h = 0)
```

This produces the plot shown in Figure 7-33.

Figure 7-32:

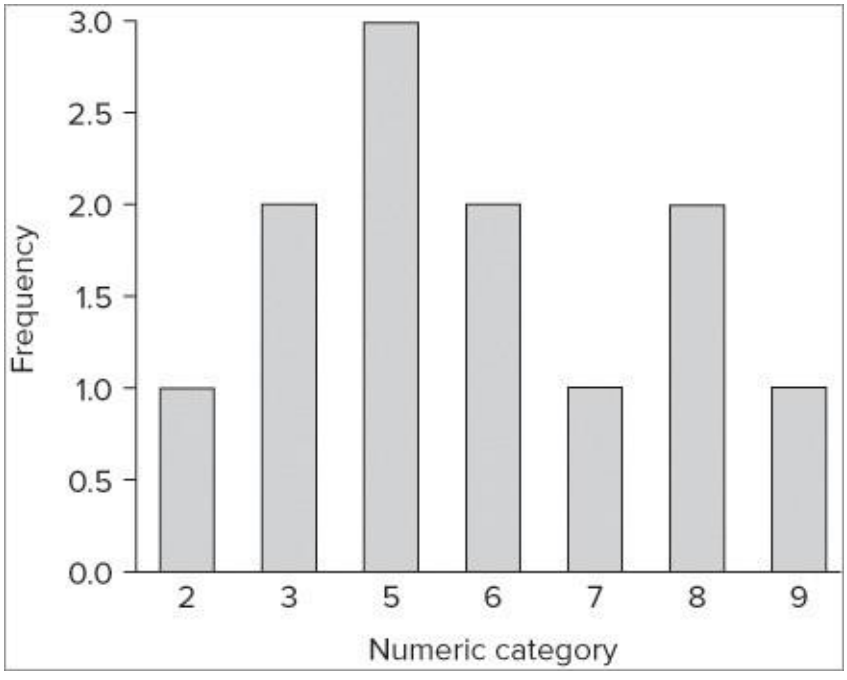
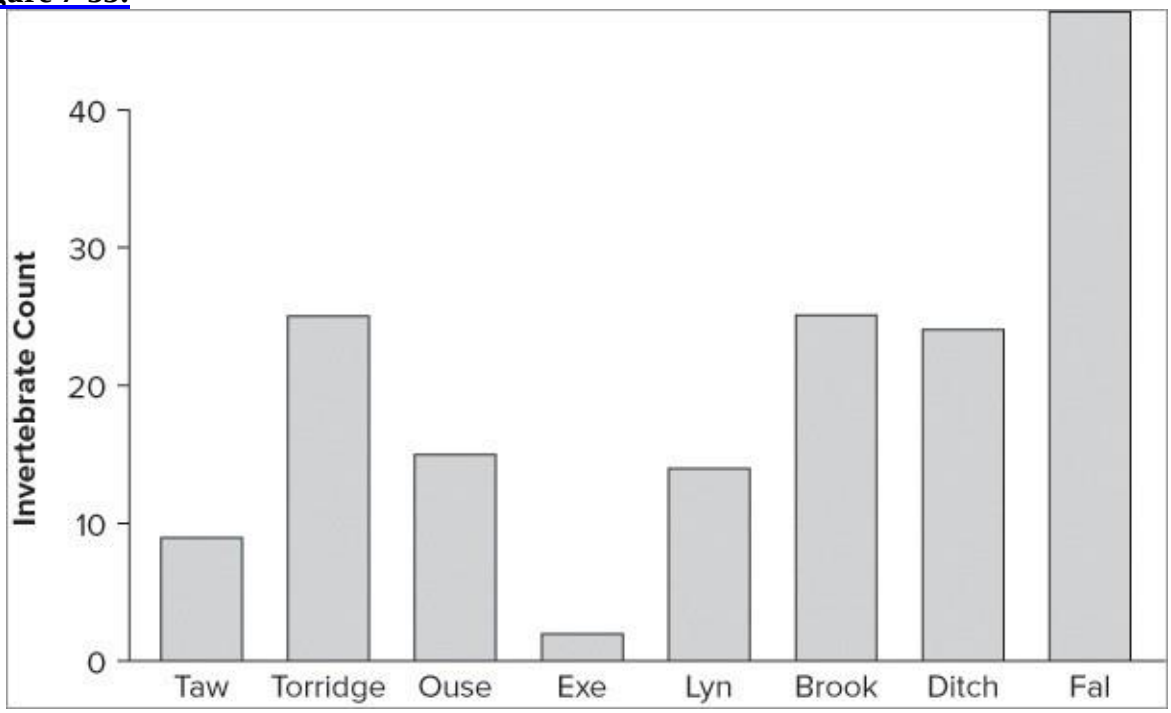


Figure 7-33:



If you try to plot the entire data frame, you get an error message:

```
> barplot(fw)
Error in barplot.default(fw) : 'height' must be a vector
or a matrix
```

This fails because you need a matrix and you only have a data frame. You need to convert the data into a matrix in some way; you can use the `as.matrix()` command to do this "on the fly" and leave the original data unchanged likeso:

```
> barplot(as.matrix(fw))
```

This is not a particularly sensible plot. If you try it you see that you get two bars, one for each column in the data. Each of these bars is a stack of several sections, each relating to a row in the data. This kind of bar chart is called a **stacked bar chart,** and you look at this in the nextsection.

Multiple Category Bar Charts

The examples of bar charts you have seen so far have all involved a single "row" of data, that is, all the data relate to categories in one group. It is also quite common to have several groups of categories. You can display these groups in several ways, the most primitive being a separate graph for each group. However, you can also arrange your bar chart so that these multiple categories are displayed on one single plot. You have two options: stacked bars and groupedbars.

Stacked Bar Charts

If your data contains several groups of categories, you can display the data in a bar chart in one of two ways. You can decide to show the bars in blocks (or groups) or you can choose to have them stacked.

The following example makes this clearer and shows a matrix data object that you have used in previous examples:

```
> 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            46       16        8       4        0
Robin             9        3        0       0        2
Song Thrush       4        0        6       0        0


> barplot(bird)
```

The plot that results is a stacked bar chart (see Figure 7-34) and each column has been split into its rowcomponents.

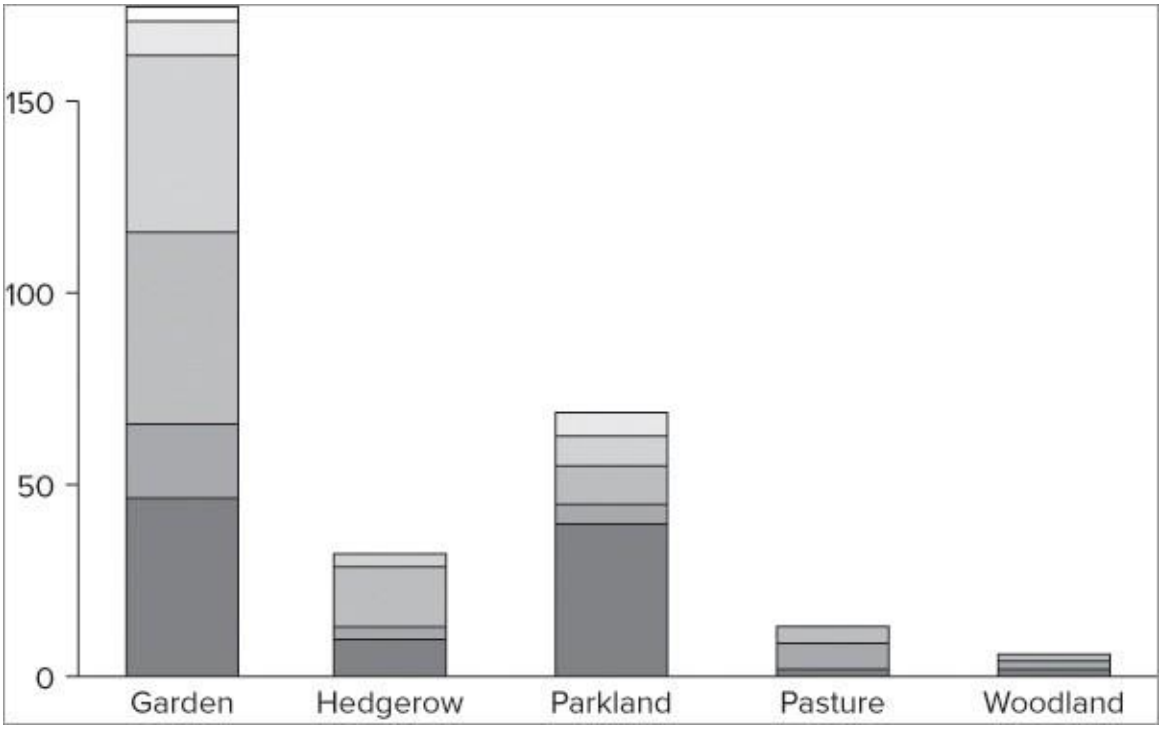
You can use any of the additional instructions that you have seen so far to modify the plot. For example, you could alter the scale of the y-axis using the `ylim =` instruction or add axis labels using the `xlab =` and `ylab =` instructions (the `title()` command can also do this). The colors shown are shades of gray,and at present there is no indication of which color belongs to which row category. You can alter this with some simple instructions, as you see in the next section.

Grouped Bar Charts

When your data are in a matrix with several rows, the default bar chart is a stacked chart as you saw in the previous section. You can force the elements of each column to be unstacked by using the `beside = TRUE` instruction as shown in the following code (the default is set to`FALSE`):

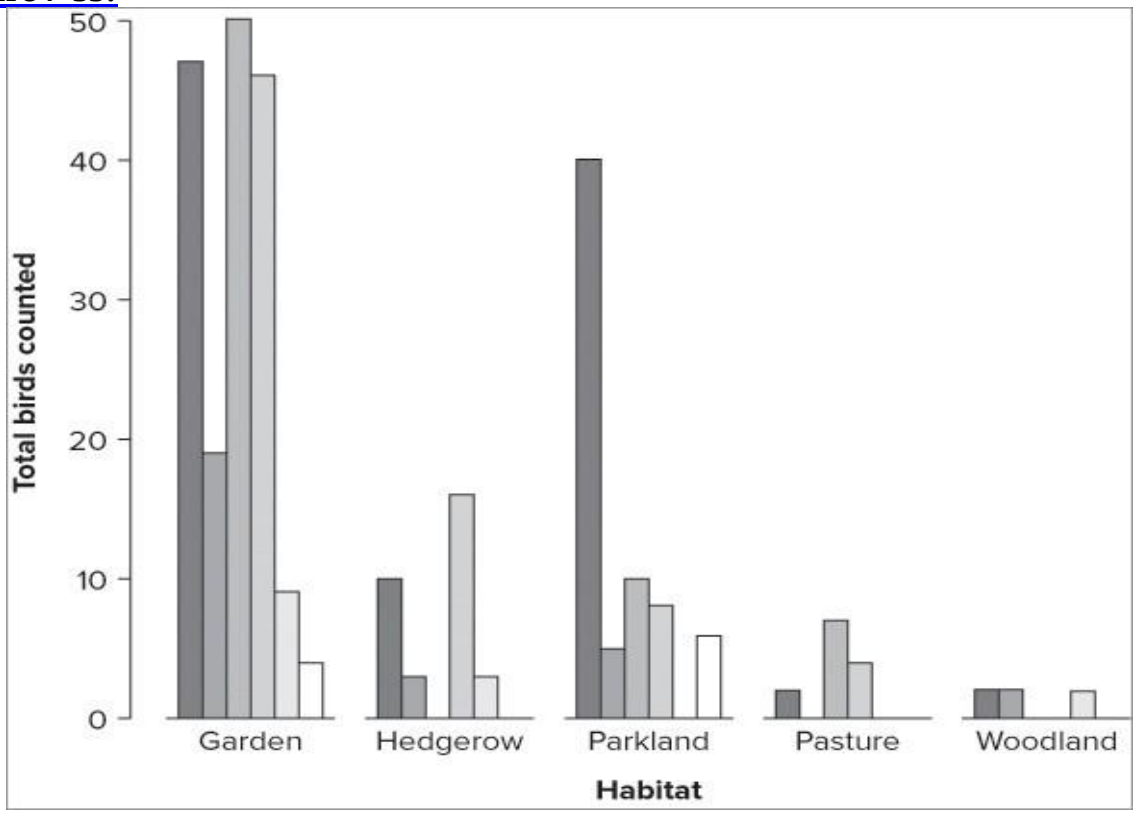
```
> barplot(bird, beside = TRUE, ylab = 'Total birds counted',
xlab
= 'Habitat')
```

Figure 7-34:



The resulting graph now shows as a series of bars in each of the column categories (Figure 7-35).

Figure 7-35:

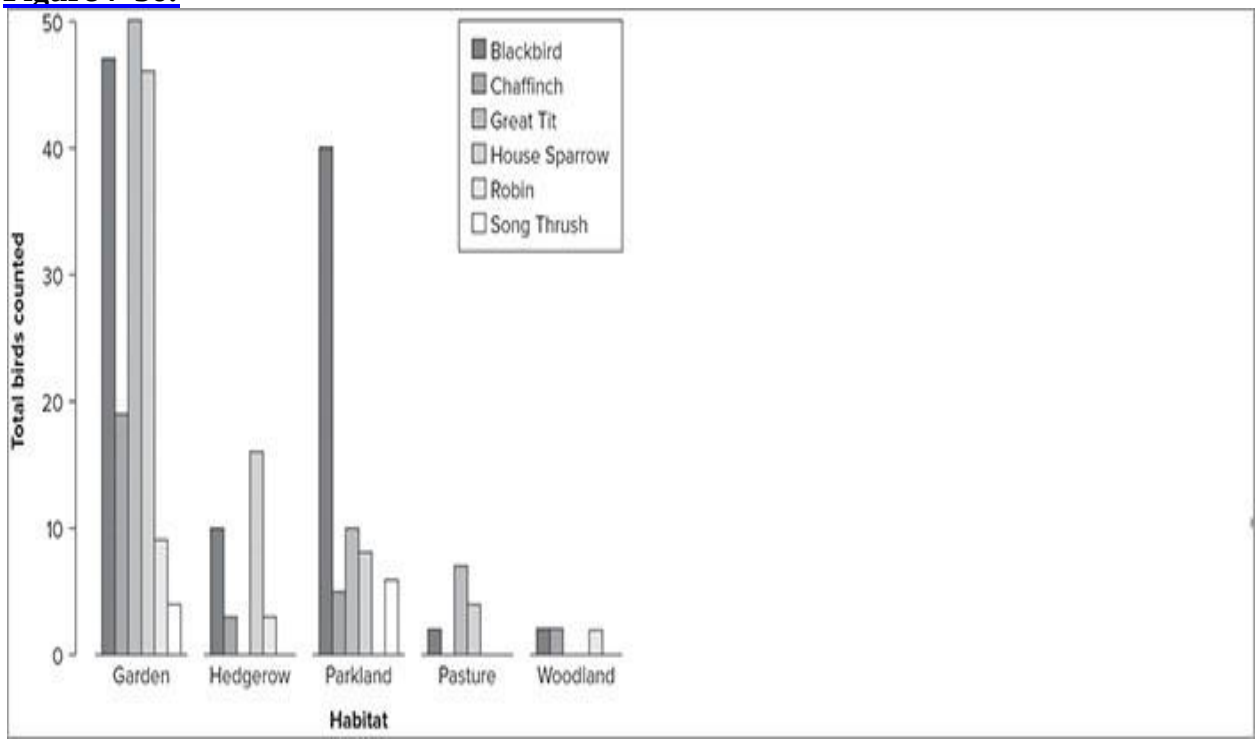


This is useful, but it is even better to see which bar relates to which row category; for this you need a legend. You can add one automatically using the `legend =` instruction, which creates a default legend that takes the colors and text from the plotitself:

```
> barplot(bird, beside = TRUE, legend = TRUE)
> title(ylab = 'Total birds counted', xlab = 'Habitat')
```

The legend appears at the top right of the plot window, so if necessary you must alter the y-axis scale using the `ylim =` instruction to get it to fit. In this case, the legend fits comfortably without any additional tweaking (see Figure 7- 36).

Figure 7-36:



You can alter the colors of the bars by supplying a vector of names in some way; you might create a separate vector or simply type the names into a `col =` instruction:

```
> barplot(bird, beside = TRUE, legend = TRUE, col =
c('black', 'pink',
'lightblue', 'tan', 'red', 'brown'))
```

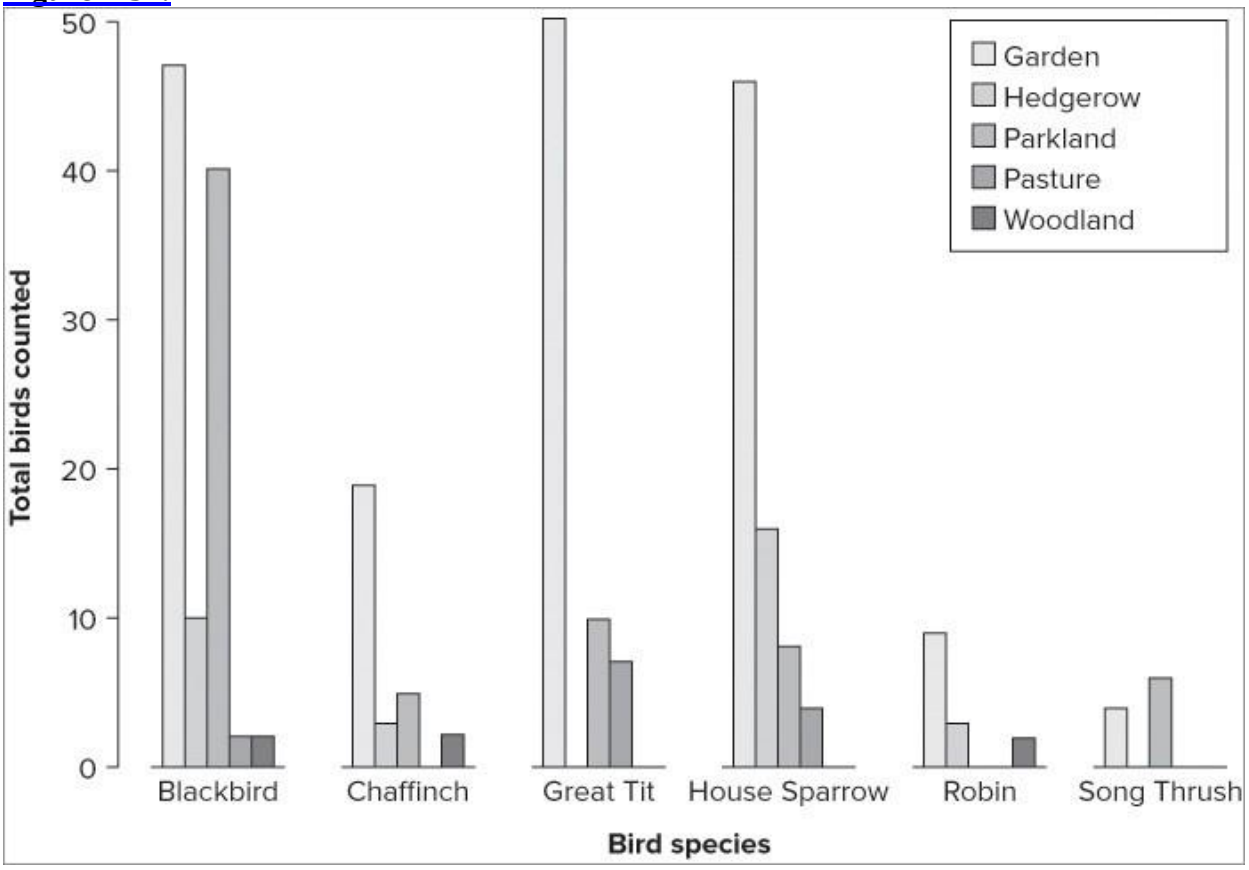
If you would rather have the row categories as the main bars, split by column, you need to rotate or transpose the matrix of data. You can use the `t()` command to do this likeso:

```
> barplot(t(bird), beside = TRUE, legend = TRUE, cex.names =
0.8, col = c('black', 'pink', 'lightblue', 'tan', 'red',
'brown'))
> title(ylab = 'Bird Count', xlab = 'Bird Species')
```

Notice this time that another instruction has been used; `cex.names = 0.8` makes the bar name labels a bit smaller so that they display neatly (see Figure 7- 37). The character expansion of the names uses a numerical value like you used previously with the `cex =` instruction; a value >1 makes text larger and <1 makes itsmaller.

So far you have seen how to create simple bar charts, and also how to make multiple category charts. It is also possible to display the bars horizontally rather than vertically, which is the subject of the next section.

Figure 7-37:



Horizontal Bars

You can make the bars horizontal rather than the default vertical using the `horiz = TRUE` instruction (this is slightly different from the instruction in the `boxplot()` command):

```
> barplot(bird, beside = TRUE, horiz = TRUE)
```

You can use all the regular instructions that you have met previously on horizontal bar charts as well, for example:

```
> bccol = c('black', 'pink', 'lightblue', 'tan', 'red',
'brown')
> barplot(bird, beside = TRUE, legend = TRUE, horiz =
TRUE, xlim = c(0, 60), col = bccol)
> title(ylab = 'Habitat', xlab = 'Bird count')
```

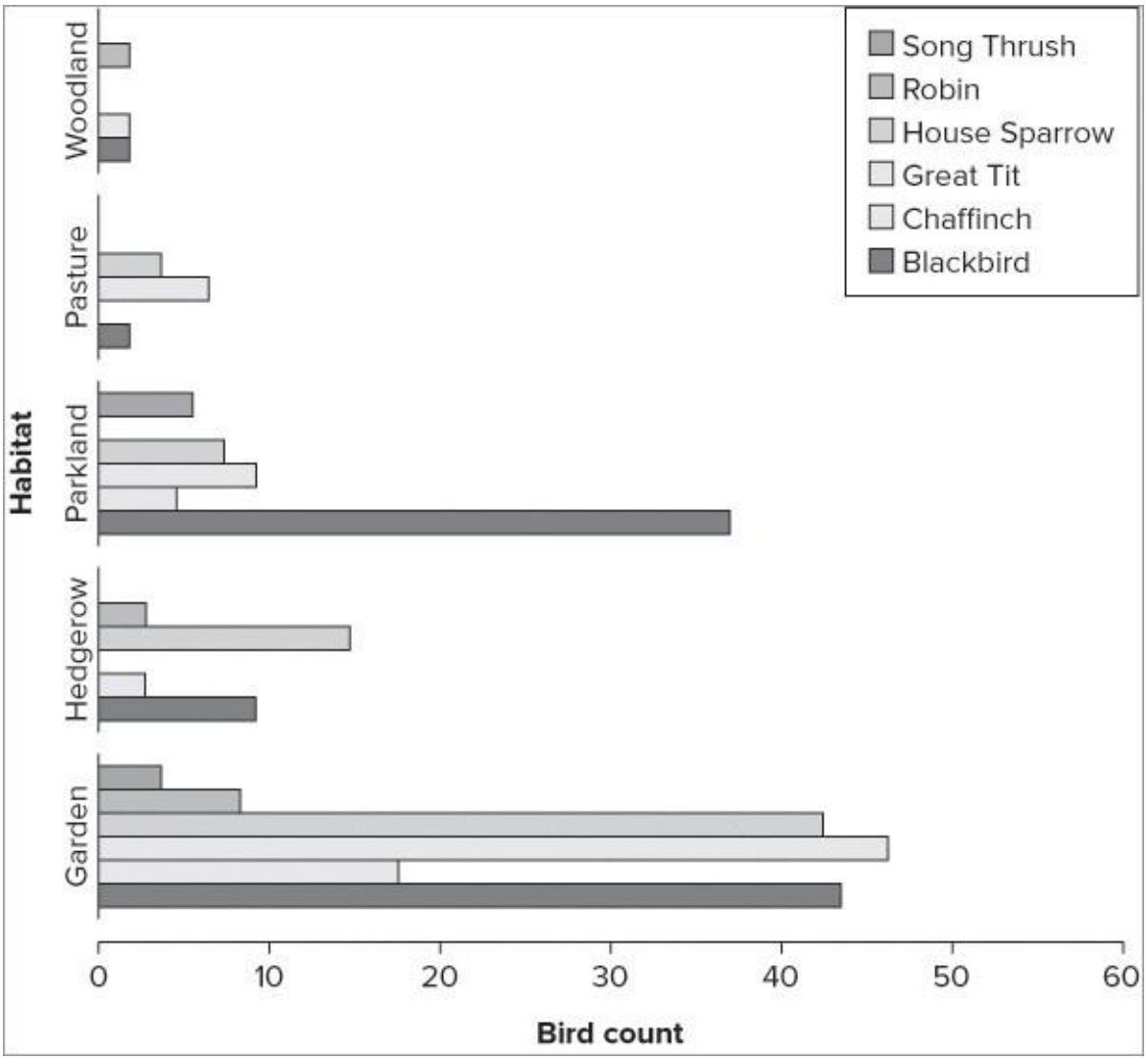
The bars now point horizontally (see Figure 7-38). The y-axis and x-axis are the original orientation as far as the commands are concerned; here the x-axis is rescaled to make it a bitlonger.

The `title()` command was used here, but you could have specified the axis labels using

`xlab =` and `ylab =` instructions as part of the `barplot()` command. In either case, you need to remember that the `xlab` instruction refers to the bottom axis and the `ylab`to the side (left) axis.

Bar Charts from SummaryData

In the examples of bar charts that you have seen so far the data were already in their final format.



However, in many cases you will have raw data that you want to summarize and plot in one go. You will most often want to calculate and present average values of various columns of a data object. You already encountered some of the commands that can produce summary results in Chapter 3; these include `colMeans()`, `apply()`, and `tapply()`.Figure 7-38:

When you have a data frame containing multiple samples, you may want to present a bar chart of their means. The following example shows a data frame

composed of several columns of numeric data; the columns are samples (that is, repeated observations). You want to summarize each one using a mean and create a bar plot of those means:

```
> head(mf)
  Length Speed Algae  NO3 BOD
1     20    12    40 2.25 200
2     21    14    45 2.15 180
3     22    12    45 1.75 135
4     23    16    80 1.95 120
5     21    20    75 1.95 110
6     20    21    65 2.75 120
```

You can make your bar plot in one of two ways. You can use the `colMeans()` command to extract the mean values, which you then plot:

```
> barplot(colMeans(mf), ylab = 'Measurement')
```

You can also use the `apply()` command to extract the mean values:

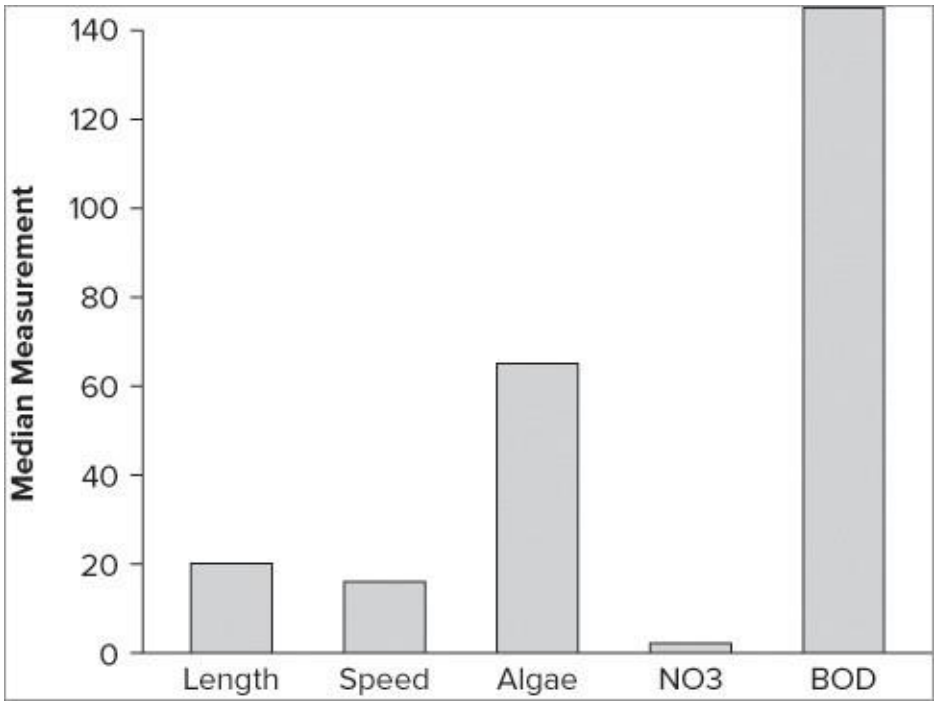
```
> barplot(apply(mf, 2, mean, na.rm = T), ylab = 'Measurement')
```

The `apply()` command is a little more complex than `colMeans()` but is also more powerful; you could chart the median values, for example:

```
> barplot(apply(mf, 2, median, na.rm = T), ylab =
'Median Measurement')
```

This produces a simple plot (see Figure 7-39).

Figure 7-39:



You can alter additional parameters to customize your `barplot()`; you return to some of these in Chapter 11. For the time being you can try the following activity to help familiarize yourself with thebasics.

Try It Out: Creating Some Bar Charts

Use the `hoglouse`, `bf`, and `bfs` data objects from the `Beginning.RData` file for this activity, which you will use to create some bar charts. The other data you will need, `VADeaths` is built in to R.

1. Look at the `hoglouse` data, which are part of the `Beginning.RData` file. The data are in a data frame, which has two columns: `fast` and `slow`. Each row is also named with the sampling site. Create a bar chart of the `fast`data as follows:

```
> barplot(hoglouse$fast, names =
rownames(hoglouse), cex.names = 0.8, col =
'slategray')
> abline(h=0)
> title(ylab = 'Hoglouse count', xlab = 'Samplingsite')
```

2. NowlookattheVADeathsdata;thesecomebuiltintoR,andyoucan view them simply by typing the name. Create a bar chart of these data. Because you have five rows you can also create five colors to helppick outthecategories.Addalegendtotheplot:

```
> cols = c('brown', 'tan', 'sienna',
'thistle','yellowgreen')
> barplot(VADeaths, legend = TRUE, col =cols)
> title(ylab = 'Death rates per 1000 peryear')
```

3. Re-plot the VADeathsdata, but this time use grouped bars:

```
> barplot(VADeaths, legend = T, beside = TRUE, col =cols)
> title(ylab = 'Death rates per 1000 peryear')
```

4. Now plot the `VADeaths` data again, but this time make the bars horizontal:

```
> barplot(VADeaths, legend = T, beside = TRUE, col = cols,
horiz =TRUE)
> title(xlab = 'Death rates per 1000 peryear')
```

5. Look at the `bf` data object. You have three columns representing three samples. Draw a bar chart of the medians for the threesamples:

```
> barplot(apply(bf, 2, median, na.rm=T), col ='lightblue')
> abline(h=0)
> title(ylab = 'Butterfly Count', xlab ='Site')
```

6. Look now at the `bfs` data object. These data are in a two-column formatwitharesponsevariableandapredictorvariable.Createanew

bar chart using the median values:

```
> barplot(tapply(bfs$count, bfs$site, FUN =
median), col = 'lightblue', xlab = 'Site', ylab =
'Butterflyabundance')
> abline(h=0)
```

How It Works

When you plot a single column from a data object, you need to use the `$` syntax to enable R to read the column. You could also use the `apply()` or `with()` commands, but in any event you need to specify the names of the data explicitly.

You can specify colors as a vector of names, which are referred to by the `col =` instruction. By default, multiple categories are represented as stacked bars; to display them as grouped bars you use `beside = TRUE` as an instruction.

If horizontal bars are required, you can use the `horizontal = TRUE` instruction. However, the left axis is still the y-axis and the bottom axis is still the x-axis.

Whenyouhavedatathatrequiresummarizingbeforeplotting,youcan embedthesummarycommandwithinthe`barplot()`command.Where dataareinmultiple-columnformat,the`apply()`commandisaflexible option,butwhenyouhavedatina`response~predictor`layoutyou usethe`tapply()`command.

Bar charts are a useful and flexible tool, as you have seen. The options covered in this section enable you to produce bar charts for a wide range of uses. You can add other instructions to the `barplot()` command, which you learn about in Chapter11.

Copy Graphics to OtherApplications

Being able to create a graphic is a useful start, but generally you need to transfer the graphs you have made to another application. You may need to make a report and want to include a graph in a word processor or presentation. You may also want to save a graph as a file on disk for later use. In this section you learn how to take the graphs you create and use them in other programs, and also how to save them as graphics files on disk.

Use Copy/Paste to CopyGraphs

When you make a graph using R, it opens in a separate window. If your graphic is not required to be of publication quality, you can use `copy` to transfer the graphic to the clipboard and then use `paste` to place it in another program. This method works for all operating systems, and the image you get depends on the size of the graphics window and the resolution of your screen.

If you need a higher quality image, or if you simply need the graphic to be saved as a graphic file to disk, the method you need to employ depends upon the operating system you areusing.

Save a Graphic to Disk

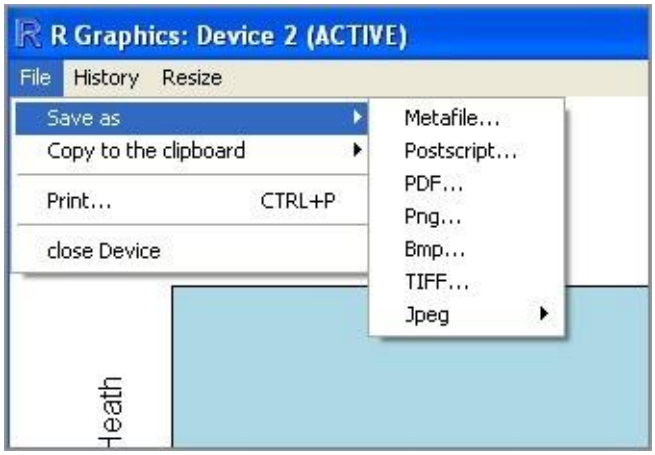
You can save a graphics window to a file in a variety of formats, including PNG, JPEG, TIFF, BMP, and PDF. The way you save your graphics depends on the operating system you are using.

In Windows and Mac the GUI has options to save graphics. In Linux you can save graphics only via direct commands, which you can also use in the other operating systems too, of course.

Windows

The Windows GUI allows you to save graphics in various file formats. Once you have created your graphic, click the graphics window and select Save As from the File menu. You have several options to choose from (see Figure7-40).

Figure 7-40:



The JPEG option gives you the opportunity to select from one of several compressions. The TIFF option produces the largest files because no compression is used. The PNG option is useful because the PNG file format is widely used and file sizes are quite small.

You can also use commands typed from the keyboard to save graphics files to disk, and you can go about this in several ways. The simplest is via the `dev.copy()` command. This command copies the contents of the graphics window to a file; you designate the type of file and the filename. To finish the process you type the `dev.off()` command. In the following example the graphics window is saved using the `png`option:

```
> dev.copy(png, file = 'R graphic
test.eps') png:R graphic test.eps
                        3
>
dev.off()
windows
      2
```

When you use this command the filename has to be specified in quotes and the file extension needs to be given. By default the file is saved in your working directory. You can see what the current working directory is using the `getwd()` command that you met in Chapter 2, “Starting Out: becoming Familiar With R.”

Macintosh

The Macintosh GUI allows graphics to be saved as PDF files. PDF is handled easily by Mac and is seen as a good option because PDF graphics can be easily rescaled. To save the

graphics window, click the window to select it and then choose Save or Save As from the File menu.

If you want to save your graphic in another format, you need to use the `dev.copy()` and `dev.off()` commands, much like you saw in the preceding section regarding the Windows operating system.

In the following example, the graphics window is saved as a PDF file. The filename must be specified explicitly—the default location for saved files is the current working directory.

```
> dev.copy(pdf, file = 'Rplot
eg.pdf') pdf
  3
>
dev.off()
quartz
     2
```

Youcansaveafiletoanotherlocationbyspecifyingthefullpathaspartofthe filename.

Linux

In Linux, R is run via the terminal and you cannot save a graphics file using "point and click" options. To save a graphics file you need to use commands typed from the keyboard.

The `dev.copy()` and `dev.off()` commands are used in the same way as described for Windows or Mac operating systems. You start by creating the graphic you require and then use the `dev.copy()` command to write the file to disk in the format you want. The process is completed by typing the `dev.off()` command.

In the following example, the graphics window is saved as a PNG file. The file is saved to the default working directory—if you want it to go somewhere else, you need to specify the path in full as part of thefilename.

```
> dev.copy(png, file = 'R graphic
test.eps') png
  3
>
dev.off()
X11cairo
       2
```

The `dev.copy()` command requires you to specify the type of graphic file you require and the filename. You can specify other options to alter the size of the final image. The most basic options are summarized in Table 7-4.

Table 7-4: Additional Graphics Instructions for the dev.copy() Command

Instruction	Explanation
`width =`	The width of the plot in pixels; defaults to 480.
`height =`	The height of the plot in pixels; defaults to 480.
`res = NA`	Theresolutioninpixelsperinch.Effectively,thedefaultworksouttobe
`quality =`	The compression to use for JPEG.

The additional instructions shown in Table 7-4 enable you to alter the size of the final graphic. The `dev.copy()` command enables you to make a graphics file quickly, but the final results are not always exactly the same as you may see on the screen. If you use a high resolution and large size, the text labels may appear as a different size from your graphics window. You find out more about graphics and the finer control of various elements of your plots in Chapter 11.

Summary

- R has extensive graphical capabilities. The basic graphs that can be produced can be customized extensively.
- The box-whisker plot is produced using the `boxplot()` command. This kind of graph is especially useful for comparing samples.
- The scatter plot is created using the `plot()` command and is used to compare two continuous variables. The `plot()` command also enables you to do this for multiple pairs of variables.
- The `plot()` command can produce line plots and the `axis()` command can create customized axes from categorical variables.
- Pie charts are used to display proportional data via the `pie()` command.
- The Cleveland dot chart is a recommended alternative to a pie chart and is created using the `dotchart()` command.

Exercises

You can find the answers to the exercises in Appendix A.

Use the `bfs` data object from the `Beginning.RData` file for Exercise 5. The other data objects are all built into R.

1. Look at the `warpbreaks` data that comes built into R. Create a box-whisker plot of the number of `breaks` for the different `tension`. Make the plot using horizontal bars and display the whiskers to the extreme range of the data. How can you draw this graph to display only a single type of wool?

2. The `trees` data come as part of R. The data is composed of three columns: the `Girth` of black cherry trees (in inches), the `Height` (in feet), and the `Volume` of wood (in cubic feet). How can you make a scatter plot of girth versus volume and display a line of best-fit? Modify the axes so that the intercept is shown clearly. Use an appropriate plotting symbol and colors to help make the chart more "interesting."

3. The `HairEyeColor` data are built into R. These data are in the form of a table, which has three dimensions. As well as the usual rows and columns, the table is split in two: `Male` and `Female`. Use the "males" table to create a Cleveland dot chart of the data. Use the mean values for the columns as an additional grouping summary result.

4. Look at the `HairEyeColor` data again. This time make a bar chart of the "female" data. Add a legend and make the colors reflect the different hair colors.

5. The `bfs` data object is part of the example data used in this book (you can download the entire data set from the companion website). Here you have

twocolumns,butterflyabundance(`count`)andhabitat(`site`).

6. Howcanyou drawabarchartofthemedianbutterflyabundancefromthethreesites?

What You Learned in This Chapter

Topic	Key Points
Box-whisker plots: `boxplot()`	The box-whisker plot is useful because it presents a lot of information in a compact manner. The data can be specified as separate vectors, an entire data frame(ormatrix) orusingtheformulasyntax.Youcanpresentbarsvertically
Scatter plots: `plot()`	Thebasic`plot()` commandisusedtomakescatterplots.Youcanspecifyth edata inseveralways,astwovectorsorasaformula.Ifyouspecifyanentiredatafra
Multiple correlation plots: `pairs()`	The`pairs()` commandisusedtocreatemultiplecorrelationplots,whichal so resultwhenthe`plot()` commandisusedonamultiplecolumndataobject.
Line plots and custom axes: `axis()`	The`plot()` commandproducespointsbydefault,butyoucanforcethepoin tsto "join up" or create a line-only plot using the `type =` instruction. The `axis()` commandisusedtocreatecustomizedaxes,especiallyusefulwhenyourx-
Pie charts: `pie()`	Piechartsareusedtodisplayproportionaldata.The`pie()` commandcanpro duce piecharts,whichcanbecustomizedtodisplaydatacounter-example.
Cleveland dot charts: `dotchart()`	The Cleveland dot chart is an alternative to the pie chart. It is more flexible becauseyoucanpresentmultiplecategoriesinoneplotandcanalsodisplaygr
Bar charts: `barplot()`	The bar chart is used to display data over various categories. The `barplot()` command can produce simple bar charts and can make stacked charts from multiplecategorydata.Thesedatacanalsobedisplayedwithbarsgroupedrat
Graphical instructions: `xlab ylab`	Manygraphicalinstructionscanbeappliedtoplots.Axislabelscanbespecifi ed andthescalesaltered.Theplottingsymbolsandcolorscanbechangedaswell
Colors: `colors()`	Manycolorsareavailabletocustomizegraphics.The`colors()` comma ndshows thenamesofthecolorsavailableandthe`col`instructionismostcommonly
Axis titles: `title()`	The`title()` commandcanaddtitlestoaxesaswellastothemainplot.Thisi san alternativetospecifyingthetitlesfromthemaingraphicalcommand,whichc
Lines on charts: `abline()`	The`abline()` commandcanbeusedtoaddlinestocharts.Horizontalorver tical linescanbeadded.Theotheruseistotakeresultsfromlinearmodelstodetermi ne slope and intercept, thus adding a line of best-fit. The lines can be
Marginal text: `mtext()`	The`mtext()` commandcanaddextratexttothemarginsofplots.Textc anbe addedtoanyaxisandplacedleft,right,orcentered.

Moving and saving graphics: `dev.copy()dev.off()`	Graphics windows can be copied to the clipboard and transferred to most other programs. The Windows and Mac GUI also permit the saving of graphics via a menu. Graphics can be saved to a file on disk using the `dev.copy()` command. Thiscancreatearangeofgraphicsformats(forexample:PNG,JPEG,TIFF,PDF)

Chapter 8

Formula Notation and Complex Statistics

What you will learn in this chapter:

- How to use formula notation for simple hypothesis tests How to use
- formula notation in graphics
- How to carry out analysis of variance (ANOVA) How to
- conduct post-hoc tests
- How the formula syntax can be used to define complex analytical models How to carry
- out complex ANOVA
- How to draw summary graphs of ANOVA How to
- create interaction plots

The R program has great analytical power, but so far most of the situations you have seen are fairly simple. In the real world things are usually more complicated, and you need a way to describe these more complex situations to enable R to carry out the appropriate analytical routines. R uses a special syntax to enable you to define and describe more complex situations. You have already met the ~ symbol; you used this as an alternative way to describe your data when using simple hypothesis testing (see Chapter 6, "Simple Hypothesis Testing") and also when visualizing the results graphically (see Chapter 7, "Introduction to Graphical Analysis"). This formula syntax permits more complex models to be defined, which is useful because much of the data you need to analyze is itself more complex than simply a comparison of two samples. In essence, you put the response variables on the left of the ~ and the predictor variable(s) on the right, likeso:

```
response ~ predictor.1 + predictor.2
```

Examples of Using Formula Syntax for Basic Tests

Some commands enable you to specify your data in one of two forms. When you carry out a `t.test()` command, for example, you can specify your data as two separate vectors or you can use the formula notation described in the introduction:

```
t.test(sample.1, sample.2)
t.test(response ~ predictor, data = data.name)
```

In the first example you specify two numeric vectors. If these vectors are contained in a data frame, you must "extract" them in some fashion using the `$` syntax or the `attach()` or `with()` commands. The options are shown in the following example:

```
> grass2
  mowunmow
1  12     8
2  15     9
3  17     7
4  11     9
5  15    NA
```

```
> t.test(grass2$mow, grass2$unmow)

> with(grass2, t.test(mow, unmow))

> attach(grass2)
> t.test(mow, unmow)
> detach(grass2)
```

In this case you have a simple data frame with two sample columns. If you haveyourdatainadifferentform,youcanusetheformulanotation:

```
> gras
  rich  graz
1   12   mow
2   15   mow
3   17   mow
4   11   mow
5   15   mow
6    8  unmo
7    9  unmo
8    7  unmo
9    9  unmo


> t.test(rich ~ graze, data = grass)
```

This time the following data frame contains two columns, but now the first is the response variable and the second is a grouping variable (the predictor factor) which contains exactly two levels (`mow` and `unmow`). Where you have more than two levels, you can use the `subset =` instruction to pick out the two you want to compare:

```
> summary(bfs)
     coun            site
 Min.   : 3.000   Arable: 9
 1st Qu.: 5.000   Grass :12
 Median : 8.000   Heath : 8
 Mean   : 8.414
 3rd
 Qu.:11.000
 Max.
        :21.00
 0

> t.test(count ~ site, data = bfs, subset = site %in%
c('Grass', 'Heath'))
```

In this example you see that you have a data frame with two columns; the first is numeric and is the response variable. The second column contains three different levels of the groupingvariable.

When you are looking at correlations you can use a similar approach. This time, however, you are not making a prediction about which variable is the predictor and which is the response,

but merely looking at the correlation between thetwo:

```
cor.test(vector.1, vector.2)
cor.test(~ vector.1 + vector.2, data = data.name)
```

In the first case you specify the two numeric vectors you want to correlate as individuals. In the second case you use the formula notation, but now you leave the left of the ~ blank and put the two vectors of interest on the right joined by + sign.

In the following activity you can practice using the formula notation by carrying out some basic statisticaltests.

Formula Notation inGraphics

When you need to present a graph of your results, you can use the formula syntax as an alternative to the basic notation. This makes the link between the analysis and graphical presentation a little clearer, and can also save you some typing.

You met the formula notation in regard to some graphs in Chapter 7, "Introduction to Graphical Analysis." For example, if you want to create abox- whisker plot of your `t.test()` result, you could specify the elements to plot using exactly the same notation as for running the test. Your options are as follows:

```
boxplot(vector.1, vector.2)
```

or

```
boxplot(response ~ predictor.1 + predictor.2, data = data.name)
```

In these examples the first case shows that you can specify multiple vectors to plot simply by listing them separately in the `boxplot()` command. The second example shows the formula notation where you specify the response variable to the left of the ~ and put the predictor variables to the right. In this example, the predictor variables are simply joined using the + sign, but you have a range of options as you see shortly.

In the case of a simple *x*, *y*, scatter plot you also have two options forcreating aplot:

```
plot(x.variable, y.variable)
```

or

```
plot(y.variable ~ x.variable, data = data.name)
```

Notice how the `x` variable is specified first in the first example, but in the secondcaseitis,ofcourse,thepredictorvariable,soitgoestotherightofthe~.

Whenever you carry out a statistical test, it is important that you also produce a graphical summary. In the following activity you will produce some graphs based on the statistical tests you carried out in the previous activity.

Try It Out: Use Formula Notation to Create Graphical Summaries of Stats Tests

Use the `grass` and `hog` data objects from the `Beginning.RData` file for this activity; you also use the `trees` code which is built-into R. You will use these data to practice use of the formula notation in producing summary graphs.

1. Earlier you looked at the `grass` data and carried out a t-test on these data. Now create a boxplot to illustrate theresult:

```
> boxplot(rich ~ graze, data = grass, col ='lightgreen')
```

2. Your graph needs some titles for the axes and perhaps a main title, so use the `title()` command to add them:

```
> title(ylab = 'Species Richness', xlab =
'Grazing Treatment', main = 'Species richness
and grazing')
```

3. Now look at the `hog` data. Earlier you conducted a U-test on these data. Create a boxplot to illustrate the result, and make the bars run horizontally:

```
> boxplot(count ~ site, data = hog, col = 'tan', horizontal
=TRUE)
```

4. This graph needs labeling so use the `title()` command to add titles to the axes as well as a main title:

```
> title(ylab = 'Water speed', xlab = 'Hoglouseabundance',
 main = 'Hoglouse and water speed')
```

5. The `trees` data comprise three columns of numerical data. Use the formula notation to produce a pairs plot showing the relationship between all the variables:

```
> pairs(~Girth + Height + Volume, data = trees)
```

6. The formula syntax can produce a scatter plot of two variables, so use it to compare `Girth` and `Height` from the `trees` data:

```
> plot(~ Girth + Height, data = trees)
```

7. Now use a more conventional formula to create the scatter plot. This time add custom titles to the axes and alter the plotting characters:

```
> plot(Girth ~ Height, data = trees, col =
'blue', cex = 1.2, xlab = 'Height (ft.)',
ylab = 'Girth (in.)')
```

8. Add a main title to the scatter plot using the `title()` command:

```
> title(main = 'Girth and Height in Black Cherry')
```

How It Works

The formula notation enables you to specify the layout of the graph, meaning that you do not need to use the `attach()` or `with()` commands, nor use the `$` symbol. All the regular graphics commands can be used.

When plotting a chart with bars/boxes horizontally, the `ylim` and `xlim` instructions refer to the left and bottom axes, respectively.

The scatter plot can handle the syntax in two ways. In the first case you used `~ predictor1 + predictor2`, and so on. This made a pairs plot in the first case showing all the predictors plotted against one another. When only two predictors are used you get a regular scatter plot, but notice that the first predictor becomes the x-axis and the second becomes the y-axis. The arrangement is different when you use the `response ~ predictor` syntax, when the response becomes the y-axis and the predictor becomes the x-axis. Note also that the `plot()` command takes the names of the columns and uses them to make axis titles, so you need to specify them explicitly as part of the command if you want something different.

Analysis of Variance(ANOVA)

Analysis of variance is an analytical method that allows for comparison of multiple samples. It is probably the most widely used of all statistical methods and can be very powerful. As the name suggests, the method looks at variance, comparing the variability between samples to the variability within samples. This is not a book about statistics, but the analysis of variance is such an important topic that it is important that you can carry out ANOVA.

The formula notation comes in handy especially when you want to carry out analysis of variance or linear regression, see Chapter 10, "Regression (Linear Modeling)." You are able to specify quite complex models that describe your data and carry out the analyses you require. You can think of analysis ofvariance

as a way of linear modeling. R has an `lm()` command that carries out linear modeling, including ANOVA (see Chapter 10). However, you also have a "convenience" command, `aov()`, that gives you an extra option or two that are useful for ANOVA.

One-WayANOVA

You can use the `aov()` command to carry out analysis of variance. In its simplest form you would have several samples to compare. The following example shows a simple data frame that comprises three columns of numerical data:

```
> head(bf)
  Grass Heath Arable
1     3     6     19
2     4     7      3
3     3     8      8
4     5     8      8
5     6     9      9
6    12    11     11
```

Toruntheaov() commandyoumusthaveyourdatainadifferentlayoutfrom the one in the preceding example, which has a column for each numerical sample.Withaov() yourequireoneresponsevariable(thenumericaldata)and one predictor variable that contains several levels of a factor (character labels). Toachievethisrequiredlayoutyouneedtoconvertyourdatausingthestack() command.

Stacking the Data before Running Analysis of Variance

The `stack()` command can be used in several ways, but all produce the same result; a two-column data frame. If your original data have multiple numeric vectors, you can create a stacked data frame simply by giving the name of the original data like so:

```
> stack(bf
   values    ind
1       3   Gras
2       4   Gras
```

```
3        3   Gras
4        5   Gras
5        6   Gras
6       12   Gras
...
```

This produces two columns, one called `values` and the other called `ind`. If you give your new stacked data frame a name, you can then apply new names to these columns using the `names()` command:

```
> bfs = stack(bf)
> names(bfs) = c('count',
'site')
```

However, there is a potential problem because the original data may contain `NA` items. You do not really want to keep these, so you can use the `na.omit()` command to eliminate them like so:

```
> bfs = na.omit(stack(bf))
> names(bfs) = c('count',
'site')
```

This eliminates any `NA`items and is stacked in the appropriate manner.

In most cases like this you want to keep all the samples, but you can select some of them and create a subset using the `select =` instruction as part of the `stack()` command like so:

```
> names(bf)
[1]"Grass"   "Heath"  "Arable"
> tmp = stack(bf, select = c('Grass', 'Arable'))
> summary(tmp)
     values          ind
 Min.   :3.00
        Arable:12
 1stQu.:4.00     Grass:12
 Median : 8.00
 Mean   :  8.19
 3rd
 Qu.:11.00
 Max. :21.00
 NA's   : 3.00
```

In this case you create a new item for your stacked data; you require onlytwo of the samples and so you use the `select =` instruction to name the columns required in the stacked data (note that the selected samples must be in quotes). Here you can see from the summary that you have transferred some `NA`items to the new data frame, so you should re-run the commands again but use `na.omit()` like so:

```
> tmp = na.omit(stack(bf, select = c('Grass', 'Arable')))
> summary(tmp)
     values          ind
 Min.   : 3.00   Arable: 9
```

```
 1st Qu.: 4.00   Grass :12
 Median : 8.00
 Mean   : 8.19
 3rdQu.:11.00
     Max.
        :21.0
       0
```

Running aov() Commands

Once you have your data in the appropriate layout, you can proceed with the analysis of variance using the `aov()` command. You use the formula notation to indicate which is the response variable and which is the predictor variable, like so:

```
> summary(bfs)
     coun            site
 Min.   : 3.000   Arable: 9
 1st Qu.: 5.000   Grass :12
 Median : 8.000   Heath : 8
 Mean   : 8.414
 3rd
 Qu.:11.000
 Max.
        :21.00
 0

> bfs.aov = aov(count ~ site, data = bfs)
```

In this example, you can see that you have a numeric vector as the response variable (`count`) and a predictor variable (`site`) comprising three levels (`Arable`, `Grass`, `Heath`). This kind of analysis, where you have a single predictor variable, is called one-way ANOVA. To see the result you simply type the name of the result object you created:

```
>
bfs.aov
Call:
   aov(formula = count ~ site, data = bfs)

Terms:
                    site Residuals
SumofSquares     55.3678  467.6667
Deg.ofFreedom          2        26
Residual standard error: 4.241130
Estimated effects may be
unbalanced
```

This gives you some information, but to see the classic ANOVA table of results you need to use the `summary()` command like so:

```
> summary(bfs.aov)
            Df Sum Sq Mean Sq F value Pr(>F)
```

```
site         2  55.37  27.684  1.53910.2335
Residuals   26467.67   17.987
```

Here you can now see the calculated F value and the overall significance.

Simple Post-hoc Testing

You can carry out a simple post-hoc test using the Tukey Honest Significant Difference test via the `TukeyHSD()` command, as shown in the following example:

```
> TukeyHSD(bfs.aov)
  Tukey multiple comparisons of
    means 95% family-wise
    confidence level

Fit: aov(formula = count ~ site, data = bfs)

$site
                  diff       lwr      upr     p adj
Grass-       -3.166667 -7.813821 1.480488 0.2267599
Heath-       -1.000000 -6.120915 4.120915 0.8788816
Heath-Grass   2.166667 -2.643596 6.976929 0.5110917
```

In this case you have a simple one-way ANOVA and the result of the post-hoc test shows the pair-by-pair comparisons. The result shows the difference in the means, the lower and upper 95 percent confidence intervals, and the p-value for the pairwisecomparison.

Extracting Means from aov()Models

Once you have conducted your analysis of variance and the post-hoc test, you may want to see what the mean values are for the various levels of the predictor variable. One way to do this is to use the `model.tables()` command; this shows you the means or effects for yourANOVA:

```
> model.tables(bfs.aov, type = 'effects')
Tables of

 effects

 siteArable

   Grass

   Heath 1.586 -

 1.5800.5862

rep  9.000 12.0008.0000

> model.tables(bfs.aov, type =
'means') Tables of means
Grand
```

```
mean

8.413793

 site
    Arable  GrassHeath
        10  6.833     9
rep      912.000      8
```

Thedefaultistodisplaytheeffects,soifyoudonotspecify`type='means'`, you get the effects. You can also compute the standard errors of the contrastsby using the `se = TRUE` instruction. However, this works only with a balanced design,whichthecurrentexamplelacks.

Chapter 9 looks at other ways to examine the means and other components of a complexanalysis.

In the following activity you use analysis of variance to examine some data; youwillneedtorearrangethedataintoanappropriatelayoutbeforecarryingout ANOVA,apost-hoctestandagraphicalsummary.

Two-WayANOVA

In a basic one-way ANOVA you have one response variable and one predictor variable, but you may come across a situation in which you have more than one predictor variable, as in the following example:

```
> pw
   height    plantwater
1       9vulgaris     lo
11vu    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
```

In this case you have a data frame with three columns; the first column is the response variable called `height`, and the next two columns are predictor variables called `plant` and `water`. This is a fairly simple example, but if you had more species and more treatments it becomes increasingly harder to present the data as separate samples. It is much more sensible, then, to use the layout as you see here with each column representing a certain variable. If you use the

summary() command it appears that you have a balanced experimental design likeso:

```
> summary(pw)
     height          plant
        water Min.    :5.00
        sativa   :9   hi:6
 1st Qu.: 9.50   vulgaris:9   lo :6
 Median:16.00                mid:6
 Mean   :19.44
 3rd
 Qu.:30.25
 Max. :44.00
```

The summary() command shows you that each of the predictor variables is split into equal numbers of observations (replicates). You now have to take into account two predictor variables, so your ANOVA model becomes a little more complicated. You can use one of the two following commands to carry out an analysis of variance for these data:

```
> pw.aov = aov(height ~ plant + water, data = pw)
> pw.aov = aov(height ~ plant * water, data = pw)
```

In the first command you specify the response variable to the left of the ~ and put the predictor variables to the right, separated by a + sign. This takes into

account the variability due to each factor separately. In the second command the predictor variables are separated with a * sign, and this indicates that you also want to take into account interactions between the predictor variables. You could also have written this command in the following manner:

```
> pw.aov = aov(height ~ plant + water + plant:water, data = pw)
```

The third term in the ANOVA model is plant:water, which indicates the interaction between these two predictor variables. If you run the aov() command with the interaction included, you get the following result:

```
> pw.aov = aov(height ~ plant * water, data = pw)
> summary(pw.aov)
             Df  Sum SqMeanSq    Fvalue
              Pr(>F) plant       1          14.22
              14.22              2.4615
              0.142644
water         2 2403.11 1201.56 207.9615   4.863e-10
plant:water   2  129.78   64.89  11.2308  0.001783 **
Residuals    12   69.33    5.78
~DH-
Signif.codes:   0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Again you see the "classic" ANOVA table, but now you have rows for each of the predictor variables as well as the interaction.

More about Post-hoc Testing

You can run the Tukey post-hoc test on a two-way ANOVA as you did before, using the

TukeyHSD() command like so:

```
> TukeyHSD(pw.aov)
  Tukey multiple comparisons of means
   95% family-wise confidence level

Fit: aov(formula = height ~ plant * water, data = pw)

$plant
                    diff       lwr       upr    padj
vulgaris-sativa -1.777778 -4.246624 0.6910687 0.142644

$water
            diff        lwr       upr     p adj
lo-hi  -27.666667 -31.369067 -23.96427 0.0000000
mid-hi -19.000000 -22.702401 -15.29760 0.0000000
mid-lo   8.666667   4.964266  12.36907 0.0001175
> TukeyHSD(pw.aov, which = 'water')
> TukeyHSD(pw.aov, which = c('plant', 'water'))
> TukeyHSD(pw.aov, which = 'plant:water')
```

In the first case you look for pairwise comparisons for the `water` treatment only.In the second case you look at both the `water` and `plant` factors independently. In the final case you look at the interaction term only. The results show you the difference in means and also the lower and upper confidence intervals at the 95 percent level. You can alter the confidence level using the `conf.level =`instruction like so:

```
> TukeyHSD(pw.aov, which = 'plant:water', conf.level =
  0.99) Tukey multiple comparisons of means
   99% family-wise confidence level

Fit: aov(formula = height ~ plant * water, data =
pw)

$`plant:water
`                              diff         lwr         upr     p

adj
vulgaris:hi-sativa:hi    -9.333333-17.8003408  -0.8663259
0.0048138
sativa:lo-sativa:hi     -33.666667 -42.1336741-25.1996592
0.0000000
vulgaris:lo-sativa:hi   -31.000000 -39.4670075-22.5329925
0.0000000
...
```

Now you can see the lower and upper confidence intervals displayed at the 99 percent level.

You can also alter the way that the output is displayed; if you look at the first column you see it

is called `diff`, because it is the difference in means. You can force this to assume a positive value and to take into account the increasing average in the sample by using the `ordered = TRUE` instruction. The upshot is that the results are reordered in a subtly different way. Also, the significant differences are those for which the lower end point is positive:

```
> TukeyHSD(pw.aov, which = 'plant:water', ordered =
  TRUE) Tukey multiple comparisons of means
    95% family-wise confidence level
```

```
    factor levels have been ordered

Fit: aov(formula = height ~ plant * water, data = pw)

$`plant:water`
                                diff         lwr       upr     p adj
vulgaris:lo-sativa:lo       2.666667 -3.92559686  9.258930 0.7490956
sativa:mid-sativa:lo        9.333333  2.74106981 15.925597 0.0048138
vulgaris:mid-sativa:lo     10.666667  4.07440314 17.258930 0.0016201
vulgaris:hi-sativa:lo      24.333333 17.74106981 30.925597 0.0000004
sativa:hi-sativa:lo        33.666667 27.07440314 40.258930 0.0000000
sativa:mid-vulgaris:lo      6.666667  0.07440314 13.258930 0.0469217
vulgaris:mid-               8.000000  1.40773647 14.592264 0.0149115
vulgaris:hi-vulgaris:lo    21.666667 15.07440314 28.258930 0.0000014
sativa:hi-vulgaris:lo      31.000000 24.40773647 37.592264 0.0000000
vulgaris:mid-sativa:mid     1.333333 -5.25893019  7.925597 0.9810084
vulgaris:hi-sativa:mid     15.000000  8.40773647 21.592264 0.0000684
sativa:hi-sativa:mid       24.333333 17.74106981 30.925597 0.0000004
vulgaris:hi-               13.666667  7.07440314 20.258930 0.0001702
sativa:hi-vulgaris:mid     23.000000 16.40773647 29.592264 0.0000007
sativa:hi-vulgaris:hi       9.333333  2.74106981 15.925597 0.0048138
```

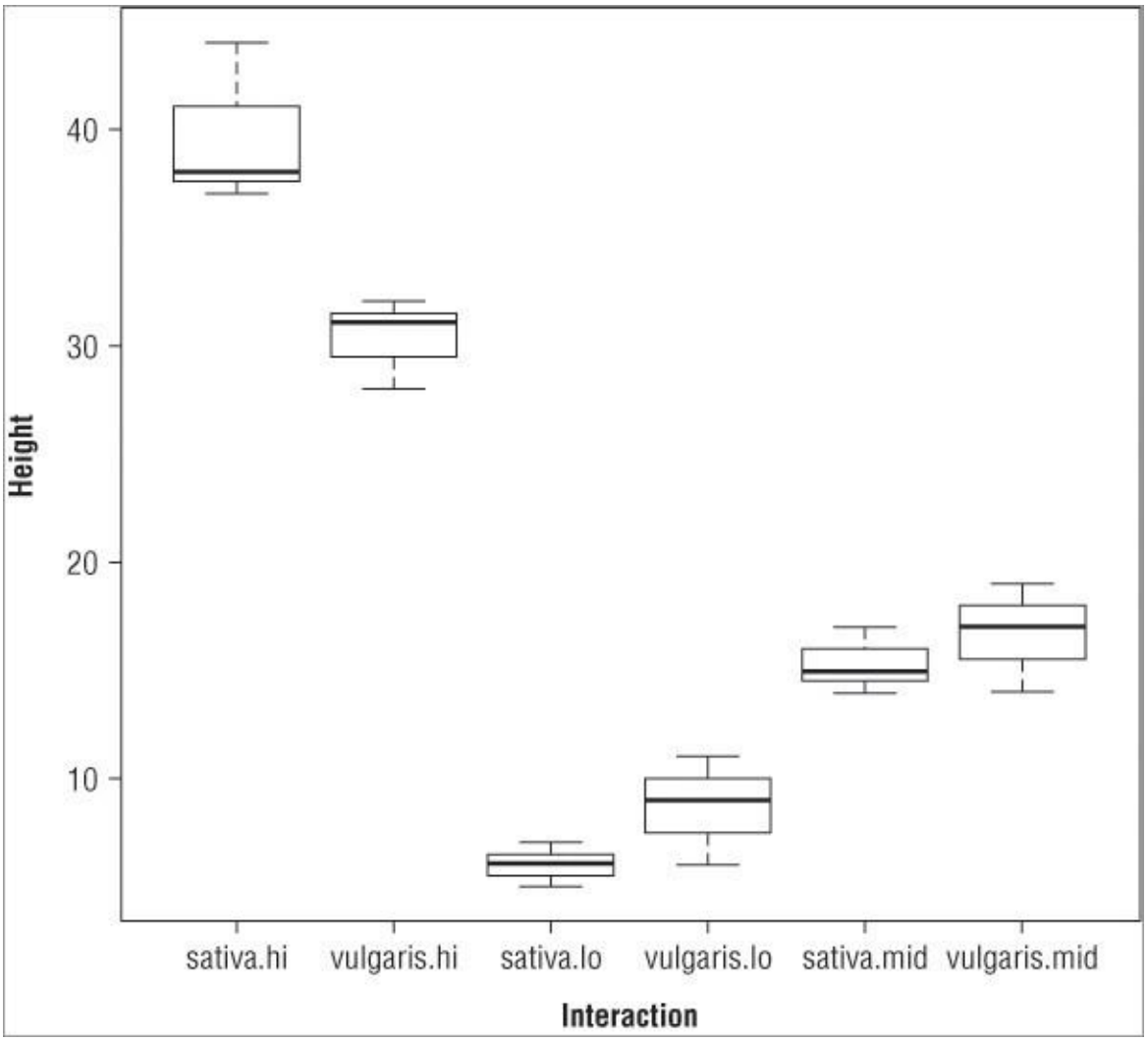
Graphical Summary of ANOVA

You should always produce a graphical summary of your analyses; a suitable graph for an ANOVA is the box-whisker plot, which you can produce using the `boxplot()` command. The instructions you give to this command to produce the plot mirror those you used to carry out the `aov()` as you can see in the following example:

```
> pw.aov = aov(height ~ plant * water, data = pw)
> boxplot(height ~ plant * water, data = pw, cex.axis = 0.9)
> title(xlab = 'Interaction', ylab = 'Height')
```

In this case an extra instruction, `cex.axis = 0.9`, is added to the `boxplot()` command. This makes the axis labels a bit smaller so they fit and display better (recall that values greater than 1 make the labels bigger and values less than 1 make them smaller). The `title()` command has also been used to add some meaningful labels to the plot, which looks like Figure8-1.

Figure8-1



If you compare the boxes to the results of the `TukeyHSD()` command, you can see that the first result listed for the interactions is `vulgaris.lo - sativa.lo`, which corresponds to the two lowest means (presented with the smaller taken away from the larger to give a positive difference in means). If you look at the second item in each pairing, you see that it corresponds to higher and higher means until the final pairing represents the comparison between the boxes with the two highest means (in this example, this is `sativa:hi –vugaris.hi`).

Graphical Summary of Post-hoc Testing

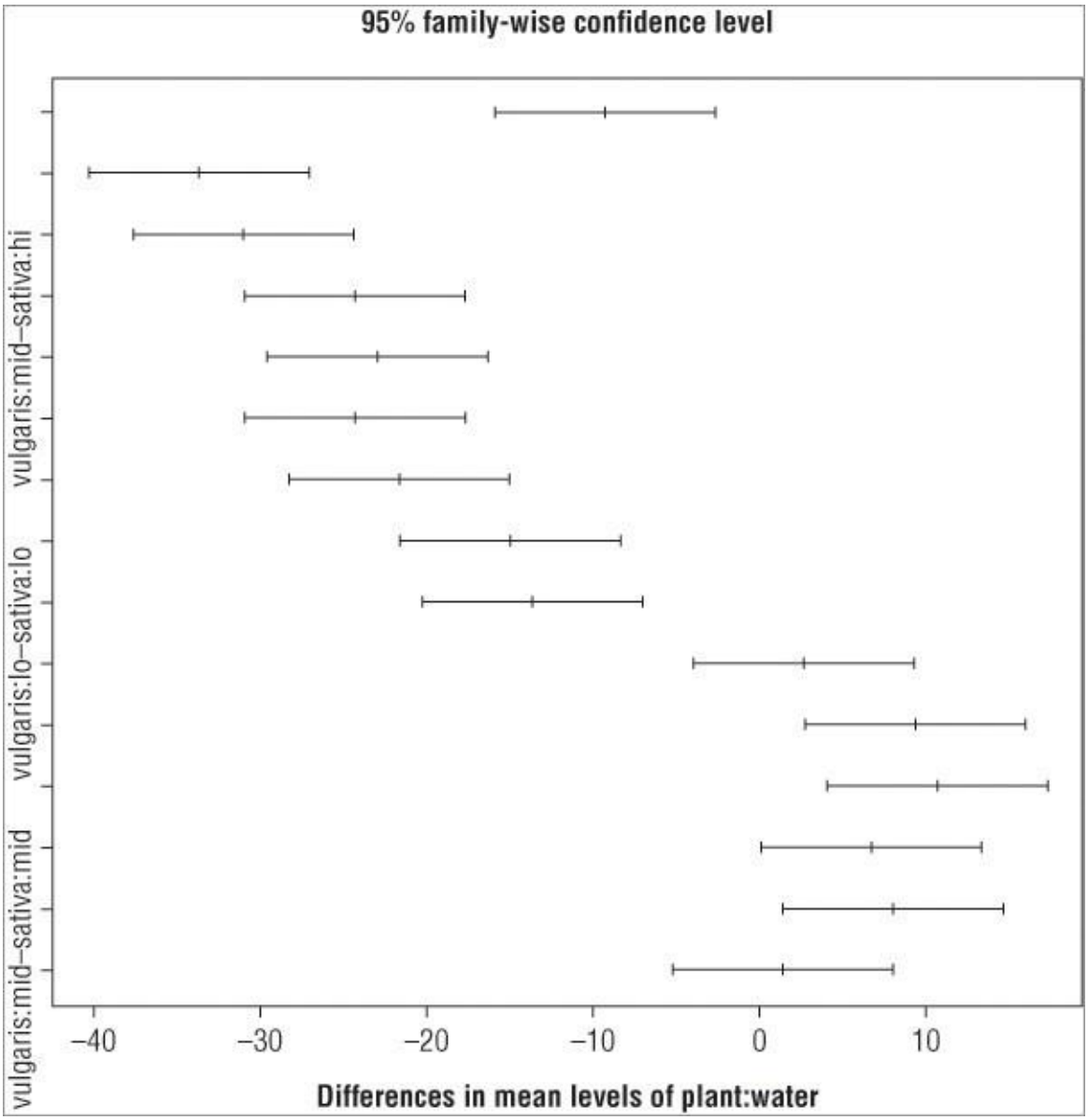
The `TukeyHSD()` command has its own dedicated plotting routine via the `plot()` command:

```
> plot(TukeyHSD(pw.aov))

> pw.ph = TukeyHDS(pw.aov)
```

```
> plot(pw.ph)
```

In the first case the `TukeyHSD()` command is called from within the `plot()` command, and in the second case the post-hoc test was given a name and the `plot()` command is called on the result object. Both give the same result (see Figure 8-2).

Figure 8-2



The main title and x-axis labels are defaults; you cannot easily alter these. You can, however, make the plot a bit more readable. At present the y-axis labels are incompletely displayed because they evidently overlap one another. You can rotate them using the `las =` instruction so that they are horizontal. You must

give a numeric value to the instruction; Table 8-1 shows the results for various values.

Table 8-1: Options for the las Instruction for Axis Labels

Comman	Explanation
`las = 0`	Labelsalwaysparalleltotheaxis(thedefau
`las = 1`	All labels horizontal
`las = 2`	Labels perpendicular to the axes
`las = 3`	All labels vertical

You can also adjust the size of the axis labels using the `cex.axis =` instruction, where you specify the expansion factor for the labels. In this case the labels will still not fit because the margin of the plot is simply too small. You can alter this, but you need to do some juggling. The commands you need to produce a finished plot are asfollows:

```
> op = par(mar = c(5, 8, 4, 2))
> plot(TukeyHSD(pw.aov, ordered = TRUE), cex.axis = 0.7, las =
1)
> abline(v = 0, lty = 2, col = 'gray40')
> par(op)
```

You cannot set the margins from within the `plot()` command directly and must use the `par()` command to set the options/parameters you require. The `par()` command is used to set many graphical parameters, and the options set remain in force for all plots. Following are the steps you need to perform to complete the plot using the preceding commands:

1. Begin by using the `par()` command to set the margins in "lines of text"; specify bottom, left, top, and right margins, respectively, and specify them all in the command. The default settings are `(5, 4, 4, 2) + 0.1`. In this case, set the left margin to 8 to give room for the labels. Note how you give a name to your setting; this enables you to return to the original values after you are done. You may need to do a bit of juggling to get the right settings for yourgraphs.

2. Now use the `plot()` command and create your post-hoc plot using the ordered values. The labels are set to be a little smaller than standard (`cex = 0.7`) and are set to horizontal (`las =1`).

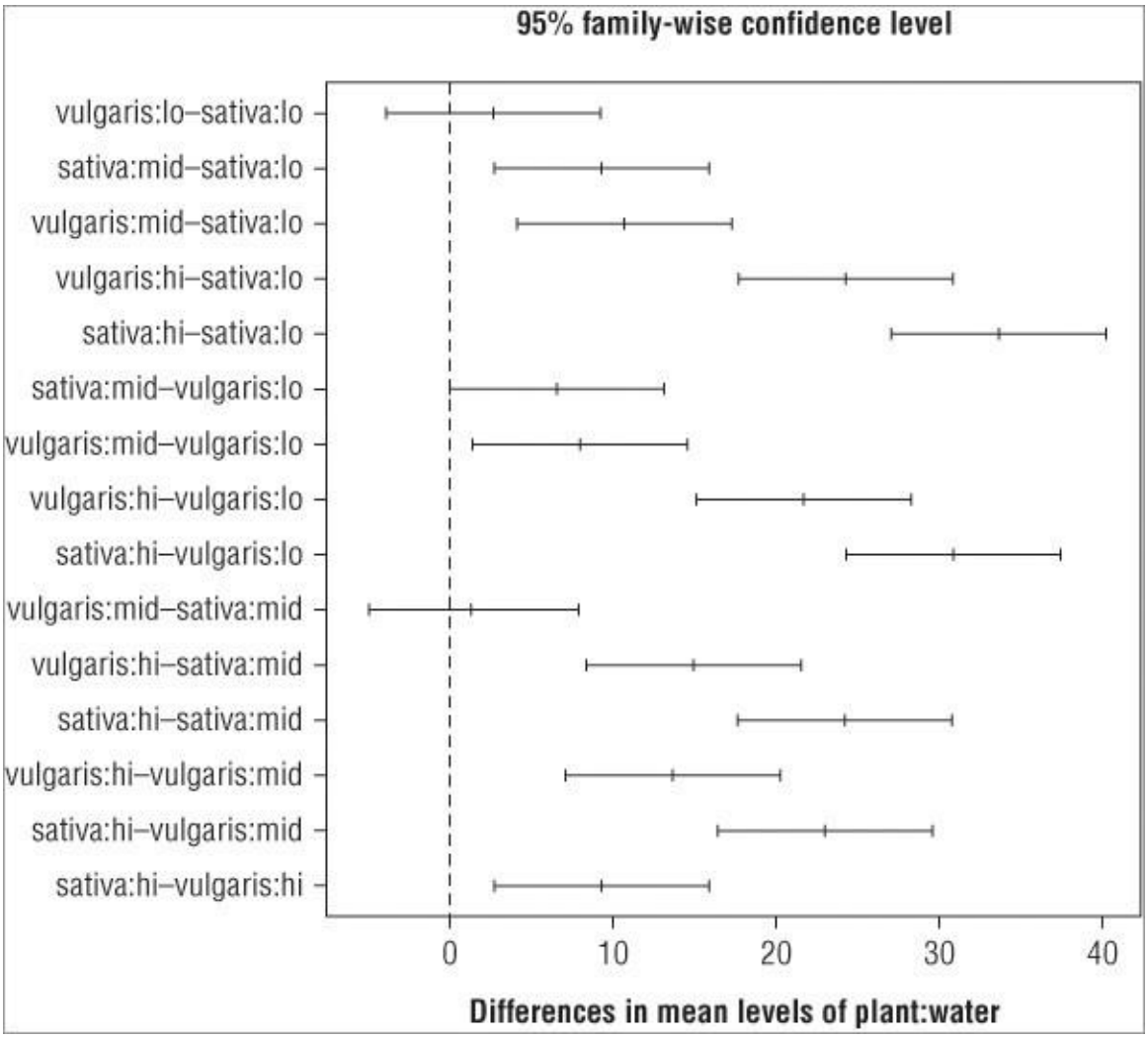
3. Next, add a vertical line using the `abline()` command. Previously you used this to create horizontal lines as well as fitting a line of best fit. Here you make a vertical line and make it dashed (`lty = 2`) and a gray color(`col = 'gray40'`).

4. The last command resets the graphical parameters to whatever they were before you altered the margin command. This perhaps seems a trifle counter- intuitive, but that is the way it is; the call to the `par()` command effectively sets the current settings and saves them to the object you named. Then it resets the graphical parameters you specified while holding the original settings in the new named object. The resulting post-hoc plot appears in Figure8-3.

By using the `order =` instruction you have ensured that all the significant pairwise comparisons have the lower end of the confidence interval > 0. This enables you to spot the

significant interactions at a glance.

Figure 8-3



In the following activity you will carry out a two-way analysis of variance using some sample data that are built-intoR.

Try It Out: Carry Out a Two-Way Analysis of Variance

In this activity you will explore the `warpbreaks` data, which come built- into R, using two-way ANOVA. You will also visualize the data as well as carrying out a post-hoc test.

1. Look at the `warpbreaks` data that come as part of R. You can use a variety of commands to explore thedata:

```
> head(warpbreaks)
> names(warpbreaks)
> str(warpbreaks)
> summary(warpbreaks)
```

2. The data contain a response variable, `breaks`, and two predictor variables, `wool` and `tension`. Create a visual summary of these data using aboxplot:

```
> boxplot(breaks ~ wool * tension, data =warpbreaks)
```

3. It looks as though some differences exist, so now carry out a two-way analysis ofvariance:

```
> wb.aov = aov(breaks ~ wool * tension, data =warpbreaks)
> summary(wb.aov)
```

4. It appears that there is a significant interaction effect. To explore the data further, carry out a Tukey post-hoctest:

```
> TukeyHSD(wb.aov)
```

5. It may be easier to see the significant differences by re-ordering the factors, so use the `order = TRUE` instruction as part of the post-hoc command:

```
> TukeyHSD(wb.aov, order = TRUE)
```

6. Visualize the post-hoc test more clearly by creating a graphical summaryoftheTukeytest:

```
> plot(TukeyHSD(wb.aov, order = T), las = 1, cex.axis =0.8)
> abline(v = 0, lty = 'dotted', col ='gray60')
```

How It Works

The`boxplot()`commandcanusetheformulasyntaxtosplitthedatainto the same chunks as you will use in the analysis of variance. The same syntax can be applied to the `aov()` command to carry out a two-way ANOVA. The summary of the analysis shows significant effects of `tension` as well as the interaction between `wool` and `tension`.

Extracting Means and SummaryStatistics

Onceyouhavecarriedoutatwo-way ANOVAanddoneapost-hoctestyouwill probably want to view the mean values for the various components of your ANOVA model. You can use a variety of commands, all discussed in the followingsections.

Model Tables

You can use the model.tables()command to extract means or effects like so:

```
> model.tables(pw.aov, type = 'means', se = TRUE) Tables of means
Grand mean

19.44444

 plant
plant
   sativa vulgaris
   20.333       18.556

 water
water
     hi       lo      mid
35.00      7.3316.00
```

```
 plant:water
                water
plant            hi        lo        mid
   sativa        39.67     6.0015.33
   vulgaris30.33            8.6716.67

Standard errors for differences of means
              plant water plant:water
              1.1331.388                 1.963
replic.           9       6                  3
```

In this case, the mean values are examined and the standard errors of the differences in the means are shown, much as you did when looking at the one- way ANOVA. In this case, because you have a balanced design, the se = TRUE instruction produce the standard errors. You see that you are shown means for the individual predictor variable as well as the interaction. If you want to see only some of the means, you can use the cterms = instruction to state what you want:

```
> model.tables(pw.aov, type = 'means', se = TRUE, cterms = c('plant:water'))
```

In this case you produce only means for the interactions and the grand mean (which you always get). If you create an object to hold the result of the model.tables()command,youcanseethatitiscomprisedofseveralelements:

```
> pw.mt = model.tables(pw.aov, type = 'means', se = TRUE)
> names(pw.mt)
[1]"tables""n"                  "se"
```

Some of these elements are themselves further subdivided:

```
> names(pw.mt$tables)
[1]"Grandmean"          "plant"          "water"          "plant:water"
```

You can therefore access any part of the result that you require using the $ syntax to slice up the result object; in the following example you extract the interactionmeans:

```
> pw.mt$tables$'plant:water'
                water
plant            hi           lo           mid sativa
                 39.66667     6.0000015.33333
   vulgaris30.33333           8.6666716.66667
```

Notice that some of the elements are in quotes for the final part; you can see this more clearly if you look at the tablespart:

```
> pw.mt$tables
$`Grand mean` [1]
19.44444

$plant
plant
   sativa vulgaris
20.33333 18.55556
```

```
$water
water
       hi        lo       mid
35.00000  7.333331 6.00000

$`plant:water`
                water
plant            hi       lo       mid sativa
                 39.66667  6.000001 5.33333
   vulgaris30.33333        8.666671 6.66667
```

The ‘Grand mean’ and ‘plant:water’ parts are in quotes. This is essentially because they are composite items—the first contains a space and the second contains a :character.

Table Commands

YoucanalsoextractcomponentsofyourANOVAmodelbyusingthetapply() command,whichyoumetpreviously,albeitbriefly(inChapter5).Thetapply() command enables you to take a data frame and apply a function to various components.Thebasicformofthecommandisasfollows:

```
tapply(X, INDEX, FUN = NULL, ...)
```

In the command, X is the variable that you want to have the function applied to;usuallythisisyourresponsevariable.YouusetheINDEXparttodescribehow you want the X variable split up. In the current example you would use the following:

```
> attach(pw)
> tapply(height, list(plant, water), FUN = mean)
                    hi        lo       mid
sativa       39.66667 6.000001 5.33333
vulgaris 30.33333 8.666667 16.66667
> detach(pw)
```

Here you use the list() command to state the variables that you require as the index (if you have only a single variable, the list() part is not needed). Note that you have to use the attach() command to enable the columns of your data frame to be readable by the command. You might also use the with() command or specify the vectors using the $ syntax; the following examples give the same result:

```
> with(pw, tapply(height, list(plant, water), FUN = mean))
> tapply(pw$height, list(pw$plant, pw$water), FUN = mean)
```

Once you have the command working you can easily modify it to use a different function; you can obtain the number of replicates, for example, by using FUN =length:

```
> with(pw, tapply(height, list(plant, water), FUN = length)) hi lo mid
sativa       3  3  3
vulgaris     3  3  3
```

Interaction Plots

It can often be useful to visualize the potential two-way interactions in an ANOVA, and indeed you probably ought to do this before actually running the analysis. You can visualize the situation using an interaction plot via the interaction.plot() command. The basic form of the command is as follows:

interaction.plot(x.factor, trace.factor, response, ...)

To obtain a basic plot, you need to specify three items:

1. Thefirstisthex.factor,whichdetermineshowtheinteractionissplit.

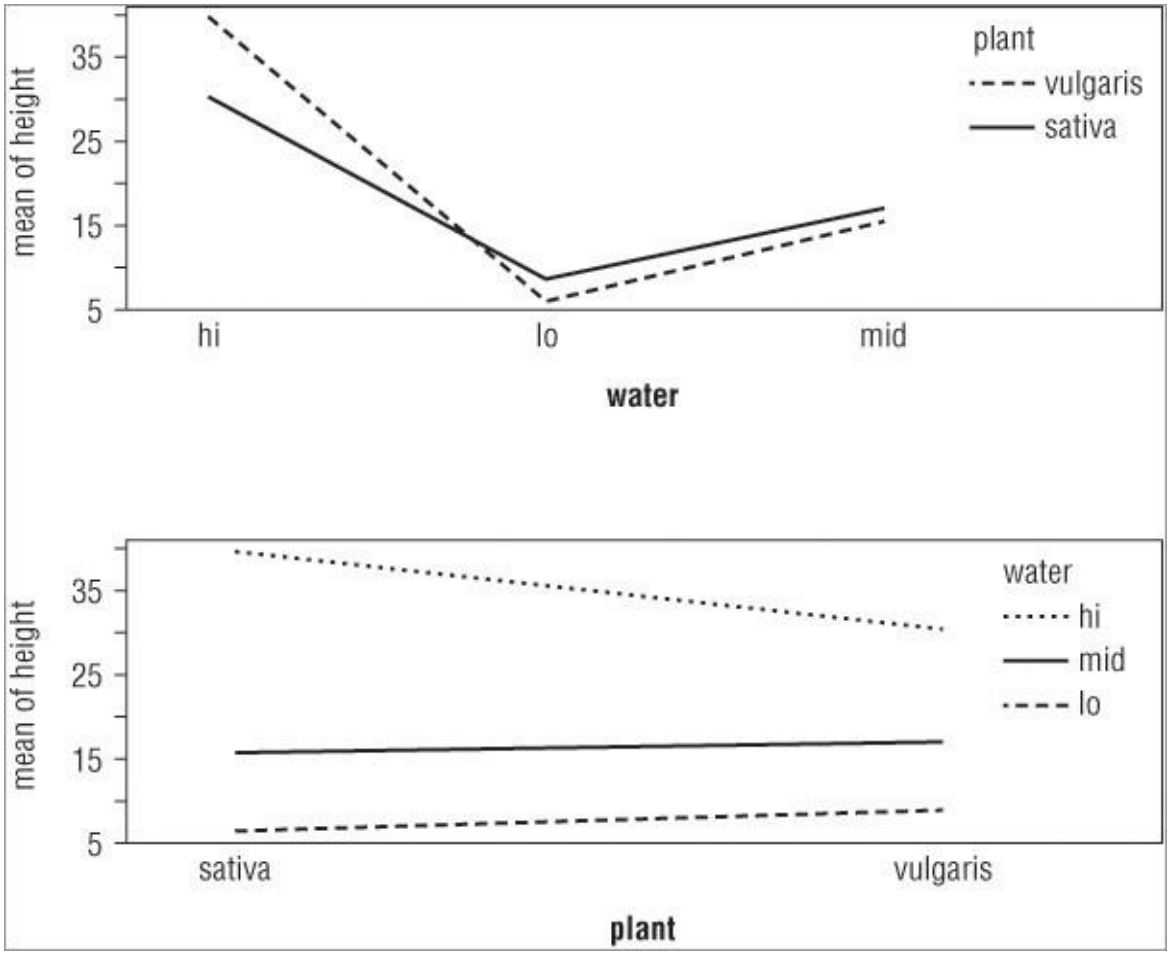
2. Then you specify the trace.factor, which is how the categories in the x.factor are split. In other words, the x.factor and the trace.factor combine to display the interaction.

3. Finally, you specify the response variable; this is plotted on the y-axis. Youcanthusaltertheappearanceoftheplotbyswappingtheorderinwhich you specify the x.factor and trace.factor variables. This is best illustratedbyshowingtwoalternativesusingthecurrentexample:

```
> interaction.plot(water, plant,height)
> interaction.plot(plant, water,height)
```

In the first case you split the x-axis by the water treatment, whereas in the second case you split the x-axis according to the plant variable. The plots that result look like Figure 8-4.

Figure 8-4



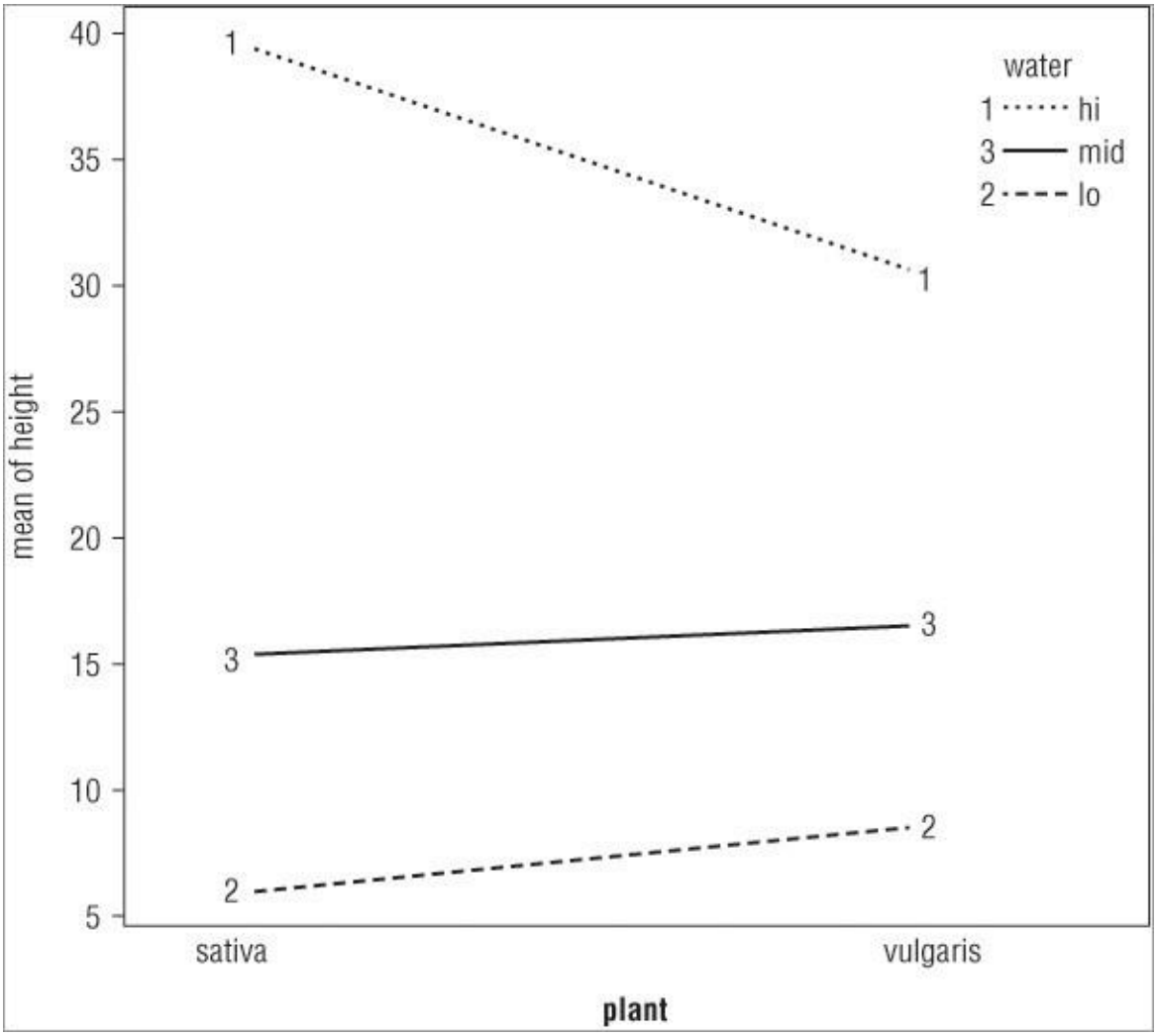
Thetopplotshowsthex-axissplitbythewatertreatmentandthebottomplot showstheaxissplitbytheotherpredictorvariable(plant).Youseehowtosplit yourplotwindowintosectionsinChapter11,"MoreAboutGraphs."

You can give a variety of additional instructions to the interaction.plot() command. To start with, you may want to see points on the plot in addition to (or instead of) the lines; you use the type = instruction to do this. You have several options: type = 'l', the default, produces lines only; type = 'b' produces lines and points; and type = 'p' shows points only. In the following example you produce an interaction plot using both lines andpoints:

```
> attach(pw)
> interaction.plot(plant, water, height, type = 'b')
> detach(pw)
```

Notice how the attach() command is used to read the variables in this case.
The plot that results looks like Figure 8-5.

Figure 8-5



When you add points, the plotting characters are by default simple numbers, but you can alter them using the pch = instruction. You can specify the pch characters in several ways, but the simplest is to use the number of levels in the

trace.factor down to 1. In this example, this would equate to pch =3:1:

```
> interaction.plot(plant, water, height, type = 'b', pch = 3:1)
```

You can also alter the style of line using the lty = instruction. The default is to use the number of levels in the trace.factor down to 1, and in the example this equates to lty = 3:1:

```
> interaction.plot(plant, water, height, type = 'b', pch = 3:1, lty = 3:1)
```

You can use colors as well as line styles to aid differentiation by using the col = instruction. By default the color is set to black, that is, col = 1, but you can specify others by giving their names or numerical values. For example:

```
> interaction.plot(plant, water, height, type = 'b', pch = 3:1,
lty = 3:1,
col = c('red', 'blue', 'darkgreen'))
```

You can also use a different summary function to the mean (which is the default) by using the

fun = instruction (note that this is in lowercase); in the following example the median is used rather than themean:

```
> interaction.plot(plant, water, height, type = 'b', pch = 3:1, fun = median)
```

You can specify new axis titles as well as an overall title by using the xlab =, ylab =, and main = instructions like you have seen previously. The following example plots lines and points using specified plotting characters and customizedtitles:

```
> interaction.plot(plant, water, height, type = 'b', pch = 3:1, xlab = 'Planttreatment',ylab
                                                    = 'Mean plant heightcm',
main = 'Interaction plot')
```

The final plot looks like Figure 8-6.

Figure 8-6

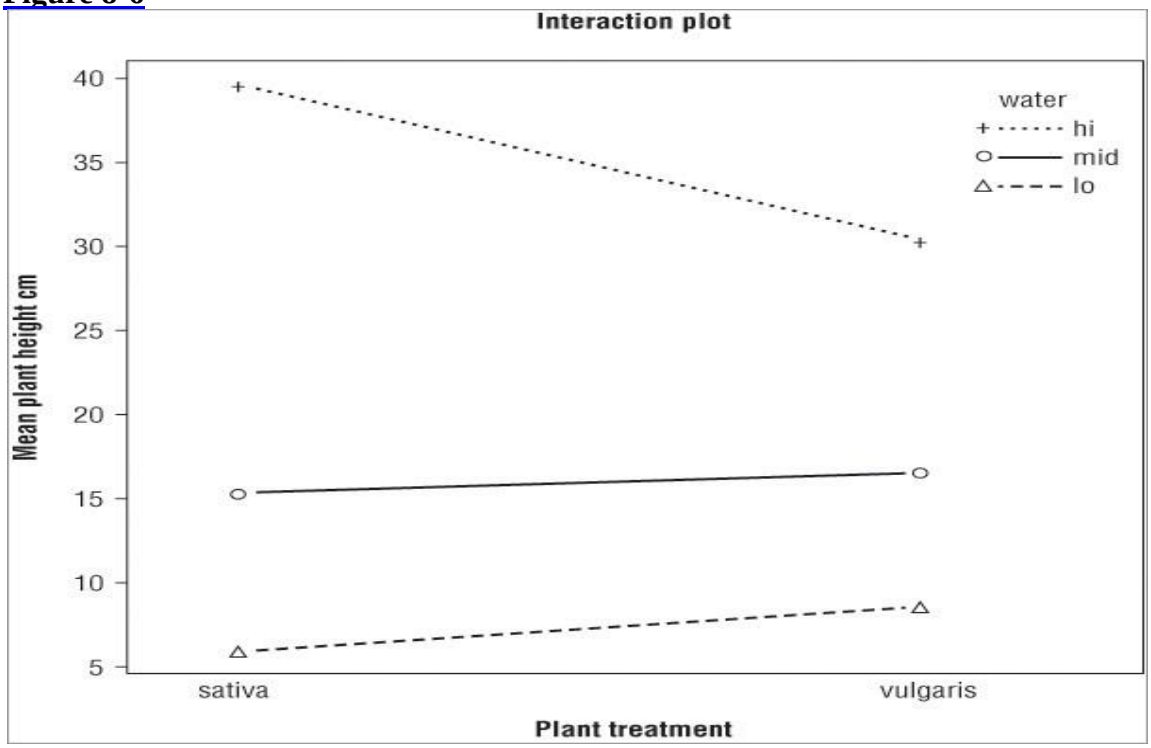


Table 8-2 shows a summary of the main instructions for the intreraction.plot()command.

Table 8-2: Instructions to be Used for the interaction.plot() Command

Instruction	**Explanation**
x.factor	The factor whose levels form the x-axis.
trace.factor	Another factor representing the interaction with the x-axis.
response	The main response factor, which is plotted on the y-axis.
fun = mean	The summary function used; defaults to the **mean**.

type = ‘l’type = ‘b’type = ‘p’	The type of plot produced: ‘l’for lines only, ‘b’for both lines and points, ‘p’
pch = as.character(1:n)pch =	Theplottingcharacters.Thedefaultusesnumericallabels,butsymbolscanbe used by specifying numeric value(s). So, pch = 1:3 produces the symbols and pch = LETTERS uses uppercase letters.
lty = nc:1	Setsthelinetype.Linetypescaneitherbespecifiedasaninteger(0=blank,1=solid[default],2=dashed,3=dotted,4=dotdash,5=longdash,6=twodash)or as one of the character strings "blank", "solid", "dashed", "dotted", "dotdash", "longdash", or "twodash", where "blank" uses “invisible lines” (thatis doesnotdrawthem)
lwd = 1	Thelinewidths.1isthestandardwidth;largervaluesmakeitwiderandsmaller valuesmakeisnarrower.
col = 1	The color of the lines. Defaults to “black”(that is, 1).
xlab = ‘x axis title’	The x-axis title.
ylab = ‘y axis title’	The y-axis title.
main = ‘main plot title’	A main graph title.
legend = TRUE	Should the legend be shown? (Defaults to TRUE.)

In the following activity you will explore a two-way ANOVA using the interaction.plot()command.

Try It Out: Make an Interaction Plot of a Two-Way ANOVA

In this activity you will revisit the warpbreaks data, which come built- into R, using the interaction.plot() command to help visualize the relationship between the variables.

1. Earlier you carried out a two-way ANOVA on the warpbreaks data. Look again at the data to remind yourself of thesituation:

```
> names(warpbreaks)
> boxplot(breaks ~ wool * tension, data =warpbreaks)
```

2. An interaction plot would be helpful to explore the relationship between tension and wool. Create a basicplot:

```
> with(warpbreaks, interaction.plot(tension, wool,breaks))
```

3. Make the plot clearer by adding data points in addition to thelines:

```
> with(warpbreaks, interaction.plot(tension, wool, breaks, type ='b'))
```

4. Now modify the plot by altering the plotting characters, colors, and linestyles:

```
> with(warpbreaks, interaction.plot(tension, wool, breaks, type = 'b', pch =
1:2, col = 1:2, lty = 1:2))
```

5. Alter the way the axis labels are displayed and force them to be horizontal:

```
> with(warpbreaks, interaction.plot(tension, wool, breaks, type = 'b', pch =
1:2, col = 1:2, lty = 1:2, las =1))
```

6. Now switch the data around to display the wool variable on the x-axis.Usethemedianasthevalueplottedonthey-axis.

```
> with(warpbreaks, interaction.plot(wool, tension, breaks, fun =median,
type = 'b', pch = 1:3, col = 1:3, lty = 1:3, las =1))
```

How It Works

The interaction.plot() command requires at least three pieces of information. The first is the variable to show on the x-axis, the second is the variable to display as the trace factor (that is, the lines/points), and the third is the value to plot on the y-axis. The default plot shows lines only and uses the mean as the function to apply to the y-axisdata.

You can display both lines and points using type = 'b', which presents the points as numbers. You can alter the plotting character using pch =. In this case you used 1:2 to display symbols relating to those values, but you could specify them in a different manner. c(16, 23), for example, would display symbols 16 and 23. The colors are altered using the col = 1:2 instruction, but you could specify them by name. The lty = 1:2 instruction altered the line styles. The las = 1 instruction forced the axis labels to display horizontally rather than parallel to the axis (las = 0 is thedefault).

Themeanisthedefaultandyoualteredthisinthefinalplotbyspecifying fun=median.Therearenowthreetraceitems(thatis,threelevelsofthe tensionvariable).

More Complex ANOVAModels

So far you have looked at fairly simple analyses of variance, one-way and two-way. You can use the formula notation to create more complex models as occasions demand. In all cases, you place your response variable to the left of the ~ and put your predictor variables to the right. In Table 8-3a range of formulae are shown that illustrate the possibilities for representing complex ANOVA models. In these examples, y is used to represent the response variable (a continuous variable) and x represents a predictor in the form of a continuous variable. Uppercase A, B, and C represent factors with discrete levels (that is, predictorvariables).

Table 8-3: Formula Syntax for Complex Models

Formula	Explanation
y ~ A	One-way analysis of variance.
y ~ A + x	Singleclassificationanalysisofcovariancemodelofy,withclasses determinedbyAandcovariatex.
y ~ A * By ~ A + B + A:B	Two-factornon-additiveanalysisofvarianceofyonfactorsAandB,thatis,
y ~ B %in% Ay ~ A/B	Nested analysis of variance with Bnested in A.

y ~ A + B %in% Ay ~ A + A:B	Nested analysis of variance with factor Aplus Bnested in A.
y ~ A * B * Cy ~ A + B + C + A:B + A:C + B:C +	Three-factorexperimentwithcompleteinteractionsbetweenfactorsA,B, andC.
y ~ (A + B + C)^2y ~ (A + B + C) * (A + B + C)y ~ A * B * C – A:B:Cy ~ A + B	Three-factorexperimentwithmodelcontainingmaineffectsandtwo-factor interactionsonly.
y ~ A * B + Error(C)	An experiment with two treatment factors, A and B, and error strata determinedbyfactorC.Forexample,asplitplotexperiment,withwhol
y ~ A + I(A + B)y ~ A + I(A^2)	The I() insulates the contents from the formula meaning and allows mathematical operations. In the first example, you have an additive two- way analysis of variance with A and the sum of A and B. In the second exampleyouhaveapolynomialanalysisofvariancewithAandthesq

You can see that when you have more complex models, you often have more than one way to write the model formula. The operators +, *, –, :, and ^ have explicit meanings in the formula syntax; the + simply adds new variables to the model. The * is used to imply interactions and the – removes items from the model. The : is used to show interactions explicitly, and the ^ is used to specify the level of interactions in a more generalmanner.

If you want to use a mathematical function in your model, you can use the I() command to "insulate" the part you are applying the math to. Thus y ~ A + I(B +C)isatwo-wayANOVAwithAasonepredictorvariable;theotherpredictor ismadebyaddingBandC.Thatisnotaverysensiblemodel,butthepointisthat the I() part differentiates between parts of the model syntax and regular math syntax.

You can specify an explicit error term by using the Error() command. Inside the brackets you specify terms that define the error structure of your model (for example, in repeated measures analyses).

You can use transformations and other mathematical manipulations on the data through the model itself, as long as the syntax is not interpreted as model syntax, so log(y) ~ A + log(B) is perfectly acceptable. If there is any ambiguity you can simply use the I() command to separate out the math part(s).

Other Options for aov()

A few additional commands can be useful when carrying out analysis of variance. Here you see just two, but you look at others when you come to look at linear modeling and the lm() command in Chapter 10, "Regression (Linear Modeling)."

Replications and Balance

You can check the balance in your model by using the replications() command. If you run

the command on the original data, you get something like the following:

```
> summary(pw)
      height                    plant        water
 Min.   :5.00          sativa  :9   hi:6
 1st Qu.: 9.50  vulgaris:9   lo :6
 Median:16.00                          mid:6
 Mean   :19.44  3rd
 Qu.:30.25     Max.
 :44.00

> replications(pw) plant
water
      9       6
Warning message:
In replications(pw) : non-factors ignored: height
```

In this case you have a data frame with three columns; the first is the response variable (which is numeric) and this is ignored. The next two columns are factors (character vectors) and are counted toward your replicates. In the previous example you used a two-way ANOVA, but a better way to use the command is to give the formula for the aov() model. You can do this in two ways: you can specify the complete formula or you can give the result object of your analysis. This latter option only works if you have actually run the aov() command. If you have run the aov() command, the formula is taken from the result, although you still need to state where the original data are. The following example shows the replications() command applied to the two-way ANOVA result that was calculatedearlier:

starting with a $. In the previous example you had a balanced model, and the result of the replications() command was a simple vector (it looks more complicated because it also has names). You can use this feature to make a test for imbalance because you can test to see if your result is a list ornot:

```
!is.list(replications(formula,data))
```

In short, you add !is.list() to your replications() command. This looks to see if the result is *not* a list. If it is not a list, the balance must be okay so you get TRUE as a result. If it *is* a list, there is not balance and you should get FALSE. If you run this command on the two examples, you get thefollowing:

```
> !is.list(replications(height ~ plant * water, data = pw)) [1] TRUE
> !is.list(replications(height ~ plant * water, data = pw2)) [1] FALSE
```

The first case is the original data and you have balance. In the second case the last row of data is deleted, and, of course, the balance islost.

Balance is always important in ANOVA designs. There are other ways to check for the balance and view the number of replicates in an aov() model; you look at these in the next chapter.

Summary

- The formula syntax enables you to describe your data in a logical manner, that is, with

a response variable and a series of predictor variables.

- The formula syntax can be used to carry out many simple stats tests because most of the commands will accept either formula notation or separate variables.
- Theformulasyntaxcanbeappliedtographics,whichenablesyoutoplot morecomplexarrangements.
- The aov() command carries out analysis of variance (ANOVA). The formula syntax is used to describe the analysis you require, so the data must be in the appropriate layout with columns for response and predictor variables.
- The stack() command can rearrange multiple samples into a response ~ predictor arrangement. If there are NA items, the na.omit() command can be used to remove them.
- The results of the TukeyHSD() command can be plotted graphically to help visualize the pairwise comparisons.
- The model.tables() command can be used to extract means from the data after the ANOVA has been carried out. Alternatively, you can use the tapply() command to view the means or use another function (for example, standard deviation).
- Theinteraction.plot()commandenablesyoutovisualizetheinteraction betweentwopredictorvariablesinatwo-wayANOVA.
- The replications and model balance of your data can be examined by using the replications() command.

Exercises

You can find the answers to these exercises in Appendix A.

Use the chick and bats data objects from the Beginning.RData file for these exercises.

1. Whatarethemainadvantagesoftheformulasyntax?

2. Look at the chick data object. These data represent weights of chicks fed a variety of diets. How can you prepare these data for analysis of variance? Carry out the one-way ANOVA

3. Now that you have an ANOVA result for the chick data, carry out a post- hoc test and also visualize the data and results using graphics. What steps do you need to carry out to do this?

4. Look at the bats data file. There are three bat species and two methods of counting them. The Box method involves looking in bat nest-boxes during the day, and the Det method involves using a sonic detector during the previous evening. What steps will you need to carry out to conduct an ANOVA?

5. The bats data yielded a significant interaction term in the two-way ANOVA. Look at this further. Make a graphic of the data and then followup with a post-hoc analysis. Draw a graph of the interaction.

What You Learned in This Chapter

Topic	Key Points
Formula syntax response ~ predictor	Theformulasyntaxenablesyoutospecifycomplexstatisticalmodels.Usually the responsevariablesgoontheleftandpredictorvariablesgoontheright.Thesynt

Stacking samples stack()	Inmorecomplexanalyses,thedataneedtobeinalayoutwhereeachcolumnisa separate item; that is, a column for the response variable and a column for each
Analysis of variance (ANOVA)	Theaov()commandcarriesoutANOVA.Youcanspecifyyourmodelusin gthe formula syntax and can carry out one-way, two-way, and more complicated ANOVA.
TukeyHSD()	TheTukeyHonestSignificantDifferenceisthemostcommonlyusedpost-hoctest and is used to carry out pairwise comparisons after the main ANOVA. You can plot() the result of the TukeyHSD() command to help visualize the pairwise comparisons
Interaction plots interaction.plot()	Theinteractionplotisagraphicalmeansofvisualizingthedifferenceinrespon sein a two-way ANOVA. The lines show different levels of a predictor variable (comparedtotheotherpredictor)andnon-
Extracting elements of an ANOVA model.tables() tapply()	TheelementsofanANOVAcanbeextractedinseveralways.Themodel.tables () command is able to show means or effects from an aov() result. The tapply() command is more general, but is useful in being able to use any function on the data; thus, you can extract means, standard deviation, and number of replicates from thedata
Replications and balance replications()	Thereplications()commandisaconvenientcommandthatenablesyoutoche ck thebalanceinanANOVAmodeldesign.

UNIT-IV
Terminology

Topic	Key Points
Making data items: `length()`	Vectors need to be the same length if they are to be added to a data frame or matrix. The `length()` command can query the current length or alter it. Setting the length to shorter than current truncates the item and making it longer adds `NA` items to the end.
Making data items: `names()` `row.names()` `rownames()` `colnames()`	You can use several commands to query and alter names of columns or rows. The `rownames()` and `colnames()` commands are used for matrix objects, whereas the `names()` and `row.names()` commands work on dataframes.
Stacking separate vectors: `stack()`	You can use the `stack()` command to combine vectors and so form a data frame suitable for complex analysis. This really only works when you have a single response variable and a single predictor.
Removing `NA` items: `na.omit()`	The `na.omit()` command strips out unwanted `NA` items from vectors and dataframes.
Repeated elements: `rep()` `gl()`	You can generate repeated elements, such as character labels that will form a predictor variable, by using the `rep()` or `gl()` commands. Both commands enable you to generate multiple levels of a variable based on a repeating pattern.
Factor elements: `factor()` `as.factor()` `levels()` `nlevels()` `as.numeric()`	A factor is a special sort of character variable. You can force a vector to be treated as a factor by using the `factor()` or `as.factor()` commands. The `levels()` command shows the different levels of a factor variable, and the `nlevels()` command returns the number of discrete levels of a factor variable. A factor can be returned as a list of numbers by using the `as.numeric()` command.
Constructing a data frame: `data.frame()`	Several objects can be combined to form a data frame using the `data.frame()` command.
Constructing a matrix: `matrix()` `cbind()` `rbind()`	You can create a matrix in two main ways: by assembling a vector into rows and columns using the `matrix()` command or by combining other elements. You can combine elements by column using the `cbind()` command or by row using the `rbind()` command.
Simple row or column sums or means: `rowSums()` `colSums()` `rowMeans()` `colMeans()`	Numerical objects (data frames and matrix objects) can be examined using simple row/column sums or means using `rowSums()`, `colSums()`, `rowMeans()`, or `colMeans()` commands as appropriate. These cannot take into account any grouping variable.
Simple sum using a grouping variable: `rowsum()`	The `rowsum()` command enables you to add up columns based on a grouping variable. The result is a series of rows of sums (hence the command name).
Apply a command to rows or columns: `apply()`	The `apply()` command enables you to give a command across rows or columns of a dataframe or matrix.

Use a grouping variable with any command: `tapply()`	The `tapply()` command enables the use of grouping variables and can utilize any command (for example, `mean`, `median`, `sd`), which is applied to a single vector (or element of a data frame or matrix).
Array objects: `object[x, y, z , ...]`	An array object has more than two dimensions; that is, it cannot be described simply by rows and columns. An array is typically generated by using the `tapply()` command with more than two grouping variables. The resulting array has several dimensions, each one relating to a grouping variable. The array itself can be subdivided using the square brackets and identifying the appropriate dimensions.
Summarize using a grouping variable: `aggregate()`	The `aggregate()` command can utilize any command and number of grouping variables. The result is a two-dimensional data frame; regardless of how many grouping variables are used.

Chapter 9

Manipulating Data and Extracting Components

What You Will Learn in this Chapter:

- How to create data frames and matrix objects ready for complex analyses How to create
- or set factor data
- How to add rows and columns to data objects
- How to use simple summary commands to extract column or row information
- How to extract summary statistics from complex data objects

The world can be a complicated place, and the data you have can also be correspondingly complicated. You saw in the previous chapter how to use analysis of variance (ANOVA) via the aov() command to help make sense of complicated data. This chapter builds on this knowledge by walking you through the process of creating data objects prior to carrying out a complicated analysis.

This chapter has two main themes. To start, you look at ways to create and manipulate data to produce the objects you require to carry out these complex analyses. Later in the chapter you look at methods to extract the various components of a complicated data object. You have seen some of these commands before and others are new.

Creating Data for Complex Analysis

To begin with, you need to have some data to work on. You can construct your data in a spreadsheet and have it ready for analysis in R, or you may have to construct the data from various separate elements. This section covers the latter scenario.

When you need to carry out a complex analysis, the likelihood is that you will have to make a complex data object.

You have already seen various ways to create data items: Using the

- c()command to create simple vectors
- Using the scan()command to create vectors from the keyboard, clipboard,

or a file from disk

- Usingtheread.table()commandtoreaddatapreviouslypreparedina spreadsheetorsomeotherprogram

If you read data from another application, like a spreadsheet, it is likely that your data are already in the layout you require. If you have individual vectors of data, you need to construct data frames and matrix objects before you can carry out the business of complex analysis.

Data Frames

The data frame is probably the most useful kind of data object for complex analyses because you can have columns containing a variety of data types. For example, you can have columns containing numeric data and other columns containing factor data. This is unlike a matrix where all the data must be of one type (for example, all numeric or alltext).

To make a data frame you simply use the data.frame() command, and type the names of the objects that will form the columns into the parentheses. However, you need to ensure that all the objects are of the same length. The following example contains two simple vectors of numerical data that you want to make into a data frame. They have different lengths, so you need to alter the shorter one and add NAitems to pad it out:

```
> mow ; unmow
[1] 12 15 17 11 15
[1] 8 9 7 9
> length(unmow) = length(mow)
> unmow
[1]      8     9     7     9NA
> grassy = data.frame(mow, unmow)
> grassy
    mowunmow
1     12          8
2     15          9
3     17          7
4     11          9
5     15         NA
```

The length() command is usually used to query the length of an object, but here you use it to alter the original data by setting its length to be the same as the longer item. If you use a value that turns out to be shorter than the current length, your object is truncated and the extra data are removed.

You can use a variety of other commands to set the names of the columns, and also add names for the individual rows. The following example looks at the main column names using the names()command:

```
>  names(grassy)
[1]"mow"           "unmow"
> names(grassy) = c('mown', 'unmown')
> names(grassy)
[1]"mown"          "unmown"
```

Here, you query the column names and then set them to new values. You can do something similar with row names. In the following example you create a vector of names first and then set them using the row.names()command:

```
> grn = c('Top', 'Middle', 'Lower', 'Set aside', 'Verge')
>  row.names(grassy) [1] "1" "2"
"3" "4""5"
> row.names(grassy) = grn
> row.names(grassy)
[1]"Top"                "Middle"        "Lower"         "Set aside""Verge"
```

Notice that the original row names are a simple index and appear as characters when you query them. The newly renamed data frame appears like this:

```
> grassy
           mown unmown
Top          12      8
Middle       15      9
Lower        17      7
Set aside    11      9
Verge        15     NA
```

You may prefer to have your data frame in a different layout, with one column for the response variable and one for the predictor (in most cases this is preferable). In the current example you would have one column for the numerical values, and one to hold the treatment names (mown or unmown). You can do this in several ways, depending on where youstart.

In this case you already have a data frame and can convert it using the stack() command:

```
> stack(grassy)
   values   ind
1      12  mown
2      15  mown
3      17  mown
4      11  mown
5      15  mown
6       8 unmown
7       9 unmown
8       7 unmown
9       9 unmown
10     NA unmown
```

Now you have the result you want, but you have an NA item that you do not really need. You can use na.omit()to strip out the NAitems that may occur:

```
> na.omit(stack(grassy))
   values   ind
1      12  mown
2      15  mown
3      17  mown
4      11  mown
5      15  mown
```

```
6        8 unmown
7        9 unmown
8        7 unmown
9        9 unmown
```

The column names are set to the defaults of values and ind. You can usethe names() command to alter them afterward. The stack()command really only works when you have a simple situation with all samples being related to a single predictor variable. When you need multiple columns with several predictorvariables,youneedadifferentapproach.

When you need to create vectors of treatment names you are repeating the same names over and over according to how many replicates you have. You can use the rep() command to generate repeating items and take some of the tedium out of the process. In the following example and subsequent steps, you use the rep() command to make labels to match up with the two samples you have (mow andunmow):

```
> mow ; unmow
[1] 12 15 17 11 15
[1] 8 9 7 9

> trt = c(rep('mow', length(mow)), rep('unmow', length(unmow)))
> trt
[1]"mow"  "mow"  "mow"  "mow"  "mow"  "unmow""unmow"
"unmow""unmow"

> rich = c(mow, unmow)
> data.frame(rich, trt)
   rich   trt
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
```

1. To begin, create a new object to hold your predictor variable, and use the rep() command to repeat the names for the two treatments as many times as is necessary to match the number of observations. The basic form of the rep()command is:

rep(what, times)

2. In this case you want a character name, so enclose the name in quotation marks. You could also use a numerical value for the number of repeats, but here you use the length() command to work out how many times to repeat the labels for each of the twosamples.

3. Create the final data object by joining together the response vectors as one column and the new vector of names representing the treatments (the predictor variable). The data.frame() command does the actual joining. Notice that in this example a name is not

specified for the final data frame; if you want to use the data frame for some analysis (quite likely), you should give the new frame a name like so:

```
> grass.dat = data.frame(rich, trt)
```

The rep() command is useful to help you create repeating elements (like factors) and you will see it again shortly. Before then, you look at creating matrix objects.

Matrix Objects

A matrix can be thought of as a single vector of data that is conveniently split up into rows and columns. You can make a matrix object in several ways:

- Ifyouhavevectorsofdatayoucanassembletheminrowsorcolumnsusing the rbind() or cbind()commands.
- If you have a single vector of values you can use the matrix()command.

The following examples and subsequent steps illustrate the two methods:

```
> mow ; unmow
[1] 12 15 17 11 15
[1] 8 9 7 9
> length(unmow) = length(mow)
> cbind(mow, unmow)
      mo  unmow
[1,]  12      8
[2,]  15      9
[3,]  17      7
[4,]  11      9
[5,]  15     NA
```

1. Begin with two vectors of numeric values, and because they are of unequallength,usethelength()commandtoextendtheshorterone.

2. Nextusethecbind()commandtobindtogetherthevectorsascolumnsin a matrix. If you want your vectors to be the rows, you use the rbind() command likeso:

```
> rbind(mow,unmow)
      [,1] [,2] [,3] [,4] [,5]
mow     12   15   17   11   15
unmo     8    9    7    9   NA
```

3. Notice that you end up with names for one margin in your matrix but not the other; in the first example the row names are not set, and in the second example the column names are not set. You can set the row names or column names using the rownames() or colnames()commands.

If you have your data as one single vector, you can use an alternative method to make a matrix using the matrix() command. This command takes a single vector and splits it into a matrix with the number of rows or columns that you specify. This means that your vector of data must be divisible by the number of rows or columns that you require. In the following example and subsequent steps you have a single vector of values that you use to create a matrix:

```
> rich
[1] 12 15 171115              8    9    7    9
> length(rich) = 10
```

```
> rich
 [1] 12 15 17 11 15  8  9  7  9 NA

> matrix(rich, ncol = 2) [,1] [,2]
[1,]        12        8
[2,]        15        9
[3,]        17        7
[4,]        11        9
[5,]        15       NA
```

1. Start by making sure your original data are the correct length for your matrix and, as before, use the length() command to extend it.

2. Next use the matrix() command to create a matrix with two columns. The command reads along the vector and splits it at intervals appropriate to create the columns you asked for. This has consequences for how the data finally appear; if you use the nrow = instruction to specify how many rows you require (rather than ncol), the data will not end up in their original samples because the matrix is populated column by column:

```
> mow ; unmow
[1] 12 15 17 11 15
[1]  8  9  7  9NA
> matrix(rich, nrow = 2)
     [,1] [,2] [,3] [,4] [,5]
[1,]   12   17   15    9    9
[2,]   15   11    8    7   NA
```

3. If you wish to create a matrix in rows, use the byrow = TRUEinstruction:

```
> matrix(rich, nrow = 2, byrow = TRUE)
     [,1] [,2] [,3] [,4] [,5]
[1,]   12   15   17   11   15
[2,]    8    9    7    9   NA
```

Like before with the first method, when you use the matrix() command none of the margin names are set; you need to use the rownames() or colnames() commands to setthem.

Creating and Setting Factor Data

When you create data for complex analysis, like analysis of variance, you create vectors for both the response variables and the predictor variables. The response variables are generally numeric, but the predictor variables may well be characters and refer to names of treatments. Alternatively, they may be simple numeric values with each number representing a separate treatment. When you create a data frame that contains numeric and character vectors, the character vectors are regarded as being factors. In the following example you can see a simple data frame created from a numeric vector and a character vector:

```
> rich ; graze
[1] 12 15 171115              8    9    7    9
[1]"mow"        "mow"      "mow"      "mow"      "mow"      "unmow""unmow"
"unmow" "unmow"
> grass.df = data.frame(rich, graze)
> str(grass.df)
```

'data.frame': 9obs.of 2variables:
$ rich:int 12 15 17 11 15 8 9 79
$ graze: Factor w/ 2 levels "mow","unmow": 1 1 1 1 1 2 2 2 2

When you use the str() command to examine the structure of the data frame that was created, you see that the character vector has been converted into a factor. If you add a character vector to an existing data frame, it will remain as a character vector unless you use the data.frame() command as your means of adding the new vector; you see this in amoment.

You can force a numeric or character vector to be a factor by using the factor()command:

```
> graze
[1]"mow"  "mow"  "mow"  "mow"  "mow"  "unmow""unmow"
"unmow""unmow"
> graze.f = factor(graze)
> graze.f
[1]mow  mow  mow  mow  mow  unmow unmowunmowunmow
Levels: mowunmow
```

Here you see that the original characters are made into factors, and you see the list of levels when you look at the object (note that the data are not in quotes as they were when they were a character object). If you want to add a character vector to an existing data frame and require the new vector to be a factor, you can use the as.factor() command to convert the vector to a factor. In the following example you see the result of adding a vector of characters without using as.factor() and then with the as.factor()command:

```
> grass.df$graze2 = graze
> grass.df
   rich graze graze2
1    12   mow    mow
2    15   mow    mow
3    17   mow    mow
4    11   mow    mow
5    15   mow    mow
6     8unmow  unmow
7     9unmow  unmow
8     7unmow  unmow
9     9unmow  unmow
> str(grass.df)
'data.frame': 9obs.of 3variables:
 $rich  :int  12 15 17 11 15 8 9 79
 $ graze : Factor w/ 2 levels "mow","unmow": 1 1 1 1 1 2 2 2 2
 $graze2:chr  "mow" "mow" "mow" "mow"...

> grass.df$graze2 = as.factor(graze)
> str(grass.df)
'data.frame': 9obs.of 3variables:
 $rich  :  int  12 15 17 11 15 8 9 79
```

```
$ graze :   Factor w/ 2 levels "mow","unmow":   1 1 1 1 1 2 2 2 2
$ graze2:   Factor w/ 2 levels "mow","unmow":   1 1 1 1 1 2 2 2 2
```

In the first instance you see that the character vector appears in the data frame without quotes, but the str() command reveals it is still comprised of characters. In the second case you use the as.factor() command, and the new column is successfully transferred as a factor variable. You can, of course, set a column to be a factor afterward, as you can see in the following example:

```
> grass.df$graze2 = factor(grass.df$graze2)
```

In this case you convert the graze2 column of the data frame into a factor usingthefactor()command.Ifyouusethedata.frame()commandthenany charactervectorsareconvertedtofactorsasthefollowingexampleshows:

```
> grass.df = data.frame(grass.df, graze2 = graze)
```

Notice how the name of the column created is set as part of the command; the graze2object is created on the fly and added to the data frame as a factor.

You may want to analyze how your factor vector is split up at some point because the factor vector represents the predictor variable, and shows you how many treatments are applied. You can use the levels() command to see how your factor vector is split up. You can use the command in two ways; you can useittoqueryanobjectandfindoutwhatlevelsitpossesses,oryoucanuseitto setthelevels.Followingareexamplesoftwocharactervectors:

```
> graze
[1]"mow"  "mow"  "mow"  "mow"  "mow"  "unmow""unmow"
"unmow""unmow"
> levels(graze)
NULL
```

Herethedataareplaincharactersandnolevelsareset;whenyouexaminethe datawiththelevels()commandyougetNULLasaresult.

```
> graze.f
[1]mow  mow  mow  mow  mow  unmow unmowunmowunmow
Levels: mowunmow
> levels(graze.f) [1]"mow"
                    "unmow
```

Here you see the names of the levels that you created earlier. If you have a numeric variable that represents codes for treatments, you can make the variable into a factor using the factor() command as you have already seen, but you can also assign names to the levels. In the following example you create a simple numeric vector to represent two treatments:

```
> graze.nf = c(1,1,1,1,1,2,2,2,2)
```

You can now assign names to each of the levels in the vector like so:

```
> levels(graze.nf)[1] = 'mown'
> levels(graze.nf)[2] = 'unmown'

> levels(graze.nf) [1]"mown"
```

"unmown"

```
> graze.nf
[1] 1 1 1 1 1 2 2 2 2
attr(,"levels")
[1]"mown"          "unmown"

> class(graze.nf) [1]
"numeric"
```

You can set each level to have a name; now your plain numeric values have a more meaningful label. However, the vector still remains a numeric variable rather than a factor. You can set all the labels in one command with a slight variation, as the following example shows:

```
> graze.nf = factor(c(1,1,1,1,1,2,2,2,2))
> graze.nf
[1] 1 1 1 1 1 2 2 2 2
Levels: 1 2

> levels(graze.nf) = list(mown = '1', unmown = '2')
> graze.nf
[1]mown mown mown mown mown unmown unmownunmownunmown
Levels: mownunmown
```

In this case you create your factor object directly using numeric values but wrap these in a factor() command; you can see that you get your two levels, corresponding to the two values. This time you use the levels() command to set the names by listing how you want the numbers to be replaced.

You can also apply level names to a vector as you convert it to a factor via the factor()command:

```
> graze.nf = c(1,1,1,1,1,2,2,2,2)
> graze.nf
[1] 1 1 1 1 1 2 2 2 2

> factor(graze.nf, labels = c('mown', 'unmown'))
[1]mown mown mown mown mown unmown unmownunmownunmown
Levels: mownunmown
```

In this instance you have a simple numeric vector and use the labels = instruction to apply labels to the levels as you make your factor object.

You can use the nlevels() command to give you a numeric result for the number of levels in a vector:

```
> graze
[1]"mow"  "mow"  "mow"  "mow"  "mow"  "unmow""unmow"
"unmow""unmow"
> nlevels(graze) [1] 0
```

```
> graze.f
[1]mow    mow    mow    mow    mow    unmow unmowunmowunmow
Levels: mowunmow
> nlevels(graze.f) [1] 2
```

You can also use the class() command to check what sort of object you are dealing with like so:

```
> class(graze) [1]
"character"
> class(graze.f) [1]
"factor"
```

In the first case you can see clearly that the data are characters, whereas in the second case you see that you have a factor object. The class() command is useful because, as you have seen, it is possible to apply levels to vectors of data without making them into factor objects. Take the following for example:

```
> nlevels(graze.nf) [1] 2
> class(graze.nf) [1]
"numeric"
```

In the preceding example you have set two levels to your vector, but it remains a numericobject.

If you want to examine a factor variable but only want to view the levels as numeric values rather than as characters (assuming they have been set), you can use the as.numeric() command likeso:

```
> as.numeric(graze.nf) [1] 1 1
1 1 1 2 2 2 2
```

Now you can switch between character, factor, and numeric quite easily.

Making Replicate TreatmentFactors

Youhavealreadyseenhowtocreatevectorsoflevelsusingtherep()command.
Thebasicformofthecommandis:

```
rep(what, times)
```

You can use this command to create repeating labels that you can use to create a vector of characters that will become a factor object.

```
> trt = factor(c(rep('mown', 5), rep('unmown', 4)))
> trt
[1]mown    mown    mown    mown    mown    unmown unmownunmownunmown
Levels: mownunmown
```

In this instance you make a factor object directly from five lots of mown and four lots of unmown, which correspond to the two treatments you require.

When you have a balanced design with an equal number of replications, you can use the eachinstruction like so:

```
> factor(rep(c('mown', 'unmown'), each = 5))
 [1]mown    mown    mown    mown    mown    unmownunmownunmown
```

unmownunmown
Levels: mown unmown

The each instruction repeats the elements the specified number of times. You can use the times and each instructions together to create more complicated repeatedpatterns.

You can also create factor objects using the gl() command. The general form of the commandis:

gl(n, k, length = n*k, labels = 1:n)

In this command, n is the number of levels you require and k is the number of replications for each of these levels. You can also set the overall length of the vectoryoucreateandaddspecifictextlabelstoyourtreatments.Forexample:

```
> gl(2, 5, labels = c('mown', 'unmown'))
 [1]mown mown mown mown mown unmownunmownunmown
unmownunmown
Levels: mown unmown

> gl(2, 1, 10, labels = c('mown', 'unmown'))
 [1]mown unmownmown unmownmown unmownmown unmownmown
unmown
Levels: mown unmown

> gl(2, 2, 10, labels = c('mown', 'unmown'))
 [1]mown mown unmownunmownmown mown unmownunmownmown
mown
Levels: mown unmown
```

In the first case you set two levels and require five replicates; you get five of one level and then five of the other. In the second case you set the number of replicates to 1, but also set the overall length to 10; the result is alternation between the levels until you reach the length required. In the third case you set the number of replicates to be two, and now you get two of each treatment until you reach the required length.

When you have a lot of data you will generally find it more convenient to create it in a spreadsheet and save it as a CSV file. However, for data with relativelyfewreplicatesitisusefultobeabletomakeupdataobjectsdirectlyin R. In the following activity, you practice making a fairly simple data object comprisinganumericresponsevariableandtwopredictorvariables.

Try It Out: Make a Complex Data Frame

Inthisactivityyouwillmakeadataframethatrepresentssomenumerical sample data and character predictor variables. This is the kind of thing thatyoumightanalyzeusingtheaov()command.

1. Start by creating some numerical response data. These relate to the abundanceofaplantatthreesites:

```
> higher = c(12, 15, 17, 11, 15)
> lower = c(8, 9, 7, 9)
```

> middle = c(12, 14, 17, 21, 17)

2. Now join the separate vectors to make onevariable:
> daisy = c(higher, lower, middle)

3. Make a predictor variable (the cutting regime) by creating a character vector:
> cutting = c(rep('mow', 5), rep('unmow', 4), rep('sheep',5))

4. Createasecondpredictorvariable(timeofcutting):
> time = rep(gl(2, 1, length = 5, labels = c('early', 'late')),3)[-10]

5. Assemble the dataframe:
> flwr = data.frame(daisy, cutting, time)

6. Tidyup:
>rm(higher, lower, middle, daisy, cutting, time)

7. View the finaldata:
> flwr

How It Works

You start by making the numerical response variable. In this case you have three sites and you create three vectors using the c() command; you could have used the scan() command instead. Next, you join the three vectors together. You could have done this right at the start, but this way you can see more easily that the three are different lengths.

The first predictor value (how the meadows were cut) is created as a simple character vector. You can see that you need five replicates for the first and third, but only four for the second. You use the rep() command to generate the required number of replicates.

The next predictor variable (time of year) is more difficult because each site was monitored early and late alternately. The solution is to create the alternating variable and remove the "extra." The gl() command creates the variable and is wrapped in a rep() command to make an alternating variable with length of five repeated three times. The tenth item is not required and is removed using the [-10]instruction.

Now the final data frame can be assembled using the data.frame() command,andtheunwantedpreliminaryvariablescanbetidiedaway usingtherm()command.Youcanviewthefinalresultbytypingits name:

```
> flwr
   daisycutting      time
1        12        mowearly
2        15        mow late
3        17        mowearly
4        11        mow late
5        15        mowearly
6         8      unmowearly
7         9      unmowlate
8         7      unmowearly
9         9      unmowlate
```

```
10    12   sheepearly
11    14   sheep  late
12    17   sheepearly
13    21   sheep  late
14    17   sheepearly
```

Adding Rows or Columns

When it comes to adding data to an existing data frame or matrix, you have various options. The following examples illustrate some of the ways you can add data:

```
> grassy
           mown unmown
Top          12      8
Middle       15      9
Lower        17      7
Setaside     11      9
Verge        15     NA

> grazed
[1] 11 141710       8

> grassy$grazed = grazed
> grassy
           mown unmown grazed
Top          12      8      11
Middle       15      9      14
Lower        17      7      17
Setaside     11      9      10
Verge        15     NA       8
```

In the preceding example you have a new sample and want to add this as a column to your data frame. The sample is the same length as the others so you can add it simply by using the $. In the next example you use the data.frame() command, but this time you are combining an existing data frame with a vector; this works fine as long as the new vector is the same length as the existing columns:

```
> grassy
           mown  unmown
Top          12      8
Middle       15      9
Lower        17      7
Set aside    11      9
Verge        15     NA

> grassy = data.frame(grassy, grazed)
> grassy
           mown  unmown    grazed
Top          12      8      11
```

Middle	15	9	14
Lower	17	7	17
Set aside	11	9	10
Verge	15	NA	8

Youaddarowtoadataframeusingthe[row,column]syntax.Inthefollowing example you have a new vector of values that you want to add as a row in your dataframe:

```
> Midstrip [1] 10
10 12

> grassy['Midstrip',] = Midstrip
> grassy
```

	mown	unmown	grazed
Top	12	8	11
Middle	15	9	14
Lower	17	7	17
Set aside	11	9	10
Verge	15	NA	8
Midstrip	10	10	12

You have now assigned the appropriate row of the data frame to your new vector of values; note that you give the name in the brackets usingquotes.

If the new data are longer than the original data frame, you must expand the data frame to "make room" for the new items; you can do this by assigning NA to new rows as required. In the following example you have a simple data frame and want to add a new column, but this is longer than the originaldata:

```
> grassy
```

	mown	unmow
Top	12	8
Middle	15	9
Lower	17	7
Set aside	11	9
Verge	15	NA

```
> grazed
[1] 11 14  1710  8  9

> grassy$grazed = grazed
Error in `$<-.data.frame`(`*tmp*`, "grazed", value = c(11, 14, 17, 10,  :
   replacement has 6 rows, data has 5
```

When you try to add the new data, you get an error message; there are not enoughexistingrowstoaccommodatethenewcolumn.Inthisinstancethedata framehasnamedrows;yourequireonlyoneextrarowsoyoucannametherow as you createit:

```
> grassy['Midstrip',] = NA
> grassy
```

	mown	unmown
Top	12	8
Middle	15	9

```
Lower          17        7
Set aside      11        9

Verge          15       NA
Midstrip        N       NA
                ^
> grassy$grazed = grazed
> grassy
              mown unmown grazed
Top             12       8      11
Middle          15       9      14
Lower           17       7      17
Setaside        11       9      10
Verge           15      NA       8
Midstrip        NA      NA       9
```

Once you have the additional row you can add the new column as before. In this case you added a column that required only a single additional row, but if you needed more you could do thiseasily:

```
> grassy[6:10,] = NA
> grassy
              mown unmown
Top             12       8
Middle          15       9
Lower           17       7
Setaside        11       9
Verge           15      NA
6               NA      NA
7               NA      NA
8               NA      NA
9               NA      NA
10              NA      NA
```

You added rows six to ten and set all the values to be NA. Notice, however, that the row names of the additional rows are unset and have a plain numerical index value. You have to reset the names of the rows using the row.names()command:

```
> row.names(grassy) = c(row.names(grassy)[1:6], "A", "B", "C", "D")
```

In this case you take the names from the first six rows and add to them the new names you require (in this case, uppercase letters).

When you have a matrix you can add additional rows or columns using the rbind() or cbind()commands as appropriate:

```
> grassy.m
          top upper mid lower bottom mow  12
           15    17   11    15
unmow       8     9    7     9         NA
> grazed
[1] 11 141710          8
```

```
> grassy.m = rbind(grassy.m, grazed)
> grassy.m
          top upper mid lower bottom mow  12
            15        17    11        15
unmow        8         9     7         9          NA
grazed      11        14    17        10           8

> grassy.m
        mo  unmow
 [1,]   12      8
 [2,]   15      9
 [3,]   17      7
 [4,]   11      9
 [5,]   15     NA

> grassy.m = cbind(grassy.m, grazed)
> grassy.m
        mo  unmow    grazed
 [1,]   12      8        11
 [2,]   15      9        14
 [3,]   17      7        17
 [4,]   11      9        10
 [5,]   15     NA         8
```

In the first case you use rbind() to add the extra row to the matrix, and in the second case you use cbind()to add an extra column.

You cannot use the $ syntax or square brackets to add columns or rows like you did for the data frame. If you try to add a row, for example, you get anerror:

```
> grassy.m
       mown  unmown
 [1,]    12       8
 [2,]    15       9
 [3,]    17       7
 [4,]    11       9
 [5,]    15      NA

> grassy.m[6,] = NA
Error in grassy.m[6, ] = NA : subscript out of bounds
```

You have to use the rbind() or cbind() commands to add to a matrix. You can, however, create a **blank** matrix and fill in the blanks later, as the following exampleshows:

```
> extra = matrix(nrow = 2, ncol = 2)
> extra
          [,1] [,2]
[1,]        NA   NA
[2,]        NA   NA
```

```
> rbind(grassy.m, extra) mown
      unmown
[1,]     12       8
[2,]     15       9
[3,]     17       7
[4,]     11       9
[5,]     15      NA
[6,]     NA      NA
[7,]     NA      NA
```

Here you create a blank matrix by omitting the data, which is filled in with NA items. You give the dimensions, as rows and columns, for the matrix and then use the rbind()command to add this to your existing matrix.

You can also specify the data explicitly like so:

matrix(data = NA, ncol = 2, nrow = 2) matrix(NA,
ncol = 2, nrow = 2) matrix(data = 0, ncol = 2, nrow =
2) matrix(data = 'X', ncol = 2, nrow = 2)

In the first two cases you use NA as your data, in the second case you fill the new matrix with the number zero, and in the final case you use an uppercase characterX.

Adding rows and columns of data to existing objects is useful, especially when you are dealing with fairly small data sets. You do not always want to resort to your spreadsheet for minor alterations. In the following activity you get a bit of extra practice by adding a column and then a row to a data frame you created in the previousactivity.

Summarizing Data

Summarizing data is an important element of any statistical or analytical process. However complex the statistical process is, you always need to summarize your data in terms of means or medians, and generally break up the data into more manageable chunks. In the simplest of cases you merely need to summarize rows or columns of data, but as the situation becomes more complex, you need to preparesummary information based on combinations of factors.

The summary statistics that you extract can be used to help visualize the situationortocheckreplicationandexperimentaldesign.Thestatisticscanalso beusedasthebasisforgraphicalsummariesofthedata.

You have various commands at your disposal, and this section starts with simple row/column summaries and builds toward more complex commands.

Simple Column and RowSummaries

When you only require a really simple column sum or mean, you can use the colSums() and colMeans() commands. Equivalent commands exist for the rows, too. These are all used in the followingexample:

```
> 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

```
>colMeans(fw)
  count      speed
20.125 15.000

> colSums(fw)
count speed
  161   120

> rowMeans(fw)
      TawTorridge            Ouse             Exe             Lyn           Brook           Ditch
Fal
      5.5            14.0            10.0            5.5            14.0            24.5            26.5
40.5

> rowSums(fw)
      TawTorridge            Ouse             Exe             Lyn           Brook           Ditch
Fal

81
```

28

20

11

28

49

53

In the example, the data frame has row names set so the rowMeans() and rowSums() commands show you the means and sums for the named rows. When row names are not set, you end up with a simple numeric vector as such:

```
> 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 5 5 ...

> colMeans(mf[-6])
> colMeans(mf[1:5])
> colMeans(mf[c(1,2,3,4,5)])
  Length       Speed       Algae         NO3
             BOD 19.64015.800      58.400
             2.046145.960
```

At the beginning you see that the data frame has five columns of numeric data and one character vector (actually, it is a factor). In the first case you exclude the factor column using [-6]; in the second case you specify columns one to five using [1:5]. In the last case you list all five columns you requireexplicitly.

Although these commands are useful, they are somewhat limited and are intended as **convenience** commands. They are also only useful when your data are all numeric, and you may very well have data comprising numeric predictor variables and factor response variables. In these cases you can call upon a range of other summary commands, as you seeshortly.

Complex SummaryFunctions

Whenyouhavecomplicateddatayouoftenhaveamixtureofnumericandfactor variables.ThesimplecolMeans()andcolSums()commandsarenotsufficient enoughtoextractinformationfromthesedata.Fortunately,youhaveavarietyof commandsthatyoucanusetosummarizeyourdata,andyouhaveseensomeof thesebefore.Hereyouseeanoverviewofsomeofthesemethods.

To help illustrate the options, start by taking a numeric data frame and adding a factor: a simple vector of site names:

```
> mfnames = c(rep('Taw',5), rep('Torridge',5), rep('Ouse',5), rep('Exe',5),
rep('Lyn',5))
> mf$site = factor(mfnames)

> str(mf)
'data.frame': 25obs.of                  6variables:
  $Length:int           20 21 22 23 21 20 19 16 15 14...
  $ Speed:int           12 14 12 16 20 21 17 14 16 21...
  $ Algae:int           40 45 45 80 75 65 65 65 35 30...
  $NO3       :num       2.25 2.15 1.75 1.95 1.95 2.75 1.85 1.75 1.952.35
...
```

$BOD :int 200 180 135 120 110 120 95 168 180 195...
$site : Factor w/ 5 levels "Exe","Lyn","Ouse",..: 4 4 4 4 4 55
5 5 5 ...

Now that you have a suitable practice sample, it is time to look at some of the complex summary functions that you can use.

The rowsum() Command

You can calculate the sums of rows in a data frame or matrix and group the sums according to some factor or grouping variable. In the following example, you use the rowsum() command to determine the sums for each of the sites that are listed in the sitecolumn:

> rowsum(mf[1:5], group = mf$site)

	Length	Speed	Algae	NO3	BO
Exe	88	83	235	7.15	859
Lyn	110	73	355	12.95	534
Ouse	102	76	325	10.35	753
Taw	107	74	285	10.05	745
Torridge	84	89	260	10.65	758

The result shows all the sites listed, and for each numeric variable you have the sum. Note that you specified the columns in the data using [1:5]; these are thenumericcolumns.Youcouldalsoeliminatethenon-numericcolumnlikeso:

> rowsum(mf[-6], mf$site)

In this case the sixth column contained the grouping variable. You can also specify a single column using its name (in quotes):

> rowsum(mf['Length'], mf$site) Length

Exe	88
Lyn	110
Ouse	102
Taw	107
Torridge	84

When you have a matrix, your grouping variable must be separate because a matrix is comprised of data of all the same type. In the following example, you create a simple vector specifying the groupings:

> 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

> grp = c(1,1,1,2,2,3)

```
> rowsum(bird, grp)
      Garden  Hedgerow  Parkland  Pasture  Woodland
1        116        13        55        9         4
2         55        19         8        4         2
3          4         0         6        0         0
```

The group vector must be the same length as the number of rows in your matrix; in this case, six rows of data. You might also create a character vector as in the followingexample:

```
> grp = c('black', 'color', 'color', rep('brown', 3))

> grp
[1] "black" "color" "color" "brown" "brown" "brown"

> rowsum(bird, grp)
          Garden  Hedgerow  Parkland  Pasture  Woodland
  black       47        10        40        2         2
  brown       59        19        14        4         2
  color       69         3        15        7         2
```

It is also possible to specify part of the matrix using a grouping contained within the originalmatrix:

```
> rowsum(bird[,1:4], bird[,5]) Garden Hedgerow
   Parkland Pasture
0       100        16        24       11
2        75        16        45        2
```

Here you use the last column as the grouping, and the result shows the group labels(asnumbers).However,youcanuseonlyanumericgroupingvariable,of course,becausethematrixcancontainonlydataofasingletype.

The apply() Command

You can use the apply() command to apply a function over all the rows or columns of a data frame (or matrix). To use it, you specify the rows or columns thatyourequire,whetheryouwanttoapplythefunctiontotherowsorcolumns, andfinally,theactualfunctionyouwant,likeso:

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

You replace the MARGIN part with a numeric value: 1 = rows and 2 = columns. You can also add other instructions if they are related to the function you are going to use; for example, you can exclude NA items, using na.rm = TRUE. In the following case you use the apply() command to apply the median to the first five columns of your dataframe:

```
> apply(mf[1:5], 2, median)
  Length    Speed    Algae      NO3      BOD
   20.00    16.00    65.00     1.95   145.00
```

You put the columns you require in the square brackets; in this case you used [1:5]. Because your object is a data frame, you can simply list the column names; more properly you should use [row, col] syntax:

```
> apply(mf[,1:5], 2, median)
```

Here you added the comma, saying in effect that you want all the rows but only columns one through five. If you want to apply your function to the rows, you simply switch the numeric value in the MARGINpart:

```
> apply(mf[,1:5], 1, median)
 [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
```

Notice that you have not specified MARGIN or FUN in the command, but have used a shortcut. R commands have a default order for instructions; so as longas youputtheargumentsinthedefaultorderyoudonotneedtonamethem.Ifyou do name them then the instructions can appear in any order. The full versionfor theprecedingexamplewouldbewrittenlikeso:

```
> apply(X = mf[,1:5], MARGIN = 1, FUN = median)
```

The apply() command enables you to use a wider variety of commands on rows and columns than the rowSums() or colMeans() commands, which obviouslyarelimitedtosum()andmean().However,youcanuseapply()only on entire rows or columns that are discrete samples. When you have grouping variables,youneedadifferentapproach.

Using tapply() to Summarize Using a Grouping Variable

The summary commands you have looked at so far have enabled you to look at entire rows or columns; only the rowsum() command lets you take into account a grouping variable. When you have grouping variables in the form of predictor variables, for example, you can use the tapply() command to take into account one or more factors as grouping variables.

The following illustrates a fairly simple example where you have a data frame comprising several numeric columns, and a single column that is a grouping variable (afactor):

```
> tapply(mf$Length, mf$site, FUN = sum)
   Exe    Lyn   Ouse TawTorridge
    88    110    102    107     84
```

The tapply() command works only on a single vector at a time; in this instance you choose the Length column using the $ syntax. Next you specify the INDEX that you want to use; in other words, the grouping variable. Finally, you select the function that you want to apply; here you choose the sum. The general form of the command is asfollows:

```
tapply(X, INDEX, FUN = NULL, ...)
```

If you omit the FUN part, or set it to NULL, you get a vector that relates to the INDEX. This is easiest to see in an example:

```
> tapply(mf$Length, mf$site, FUN = NULL)
 [1] 4 4 4 4 4 5 5 5 5 5 3 3 3 3 3 1 1 1 1 1 2 2 2 2 2
```

If you refer to the original data you will see that the fourth site is the Exe factor, and because this is alphabetically the first, it is returned first. The vector result shows the rows of the original data that relate to the grouping factor.

When you have more than one grouping variable, you can list several factors to be your INDEX. In the following example you have a data frame comprisinga columnofnumericdataandtwofactorcolumns:

```
> str(pw)
'data.frame': 18obs.of                  3variables:
 $height:int            9 11 6 14 17 19 28 31 32 7...
 $ plant : Factor w/ 2 levels "sativa","vulgaris": 2 2 2 2 2 2 2 2
2 1 ...
 $ water : Factor w/ 3 levels "hi","lo","mid": 2 2 2 3 3 3 1 1 1 2
...

> tapply(pw$height, list(pw$plant, pw$water), mean) hi lo       mid
sativa          39.66667 6.00000015.33333
vulgaris 30.33333 8.666667 16.66667
```

This time you specify the columns you want to use as grouping variables in a list() command; there are only two variables here and the first one becomes the rows of the result and the second becomes the columns.

If you have more than two grouping variables, the result is subdivided into more tables as required. In the following example you have an extra factor column and use all three factors as grouping variables:

```
> str(pw)
'data.frame': 18obs.of                  4variables:
 $height:int            9 11 6 14 17 19 28 31 32 7...
 $ plant : Factor w/ 2 levels "sativa","vulgaris": 2 2 2 2 2 2 2 2
2 1 ...
 $ water : Factor w/ 3 levels "hi","lo","mid": 2 2 2 3 3 3 1 1 1 2
...
 $ season: Factor w/ 2 levels "spring","summer": 1 2 2 1 2 2 1 2 2
1 ...

> pw.tap = tapply(pw$height, list(pw$plant, pw$water, pw$season), mean)
, , spring

             hi lo mid
sativa       44    7    14
vulgaris28         9    14

, , summer
```

```
                hi     lomid
sativa        37.55.5  16
vulgaris31.58.5              18
```

In this case the third grouping variable has two levels, which results in two tables,oneforspringandoneforsummer.TheresultispresentedasakindofR object called an array; this can have any number of dimensions, but in this case you have three. If you look at the structure of the result using the str() command,youcanseehowthedimensionsareset:

```
> pw.tap = tapply(pw$height, list(pw$plant, pw$water, pw$season), mean)
> str(pw.tap)
 num [1:2, 1:3, 1:2] 44 28 7 9 14 14 37.5 31.5 5.5 8.5 ...
 - attr(*, "dimnames")=List of 3
  ..$ : chr [1:2] "sativa" "vulgaris"
  ..$ : chr [1:3] "hi" "lo" "mid"
  ..$ : chr [1:2] "spring" "summer"
```

Youcanseethatthefirstdimensionisrelatedtotheplantvariable,thesecond isrelatedtothewatervariable,andthethirdisrelatedtotheseasonvariable;in other words, the dimensions are in the same order as you specified in the tapply()command.

You can use the square brackets to extract parts of your result object, but now you have the extra dimension to take into account. To extract part of the result object you need to specify three values in the square brackets (corresponding to each of the three dimensions, plant, water, and season). In the following example you select a single item from the pw.tap result object by specifying a single value for each of the three dimensions.

```
> pw.tap[1,1,1]
[1] 44
```

The item you selected corresponds to the first plant (sativa), the first water treatment (hi), and the first season (spring), and you see the result, 44. If you want to see several items you can specify multiple values for any dimension. In the following example you select two values for the plant dimension (1:2), which will display a result for both sativaand vulgaris.

```
>pw.tap[1:2,1,1]
sativavulgaris 44
               28
```

The two result values (44 and 28) correspond to the first water treatment (hi) and the first season (spring). In the following example you select multiple values for the first two dimensions (plant and water) but only a single value for the third(season).

```
> pw.tap[1:2,1:3,1]
          hi lo mid
sativa    44  7  14
vulgaris28    9  14
```

Now you can see that you have selected all of the plant and water treatments but only a single season(spring).

The result is an array object, and as you have seen, it can have multiple dimensions. You can use the class() command to determine that the result is indeed an array object:

```
> class(pw.tap) [1]
"array"
```

Summarizing data using grouping variables is an important task that you will need to undertake often. Most often you will need to check means or medians for the groups, but many other functions can be useful. In the following activity you practice using the mean, but you could try out some other functions (for example, median, sum, sd, orlength).

The tapply() command is very useful, but you may only want to have a simple table/matrix as your result rather than a complicated array. It is possible to summarize a data object using multiple grouping factors using other commands, as you see next.

The aggregate() Command

The aggregate() command enables you to compute summary statistics for subsets of a data frame or matrix; the result comes out as a single matrix rather than an array item, even with multiple grouping factors. The general form of the command is asfollows:

```
aggregate(x, by, FUN, ...)
```

You specify the data you want followed by a list() of the grouping variables and the function you want to use. In the following example you use a single grouping variable and the sum() function:

```
> aggregate(mf[1:5], by = list(mf$site), FUN = sum)
      Group.1 Length Speed Algae   NO3  BO
1         Exe     88    83   235  7.15 859
2         Lyn    110    73   355 12.95 534
3       Ouse     102    76   325 10.35 753
4         Taw    107    74   285 10.05 745
5    Torridge     84    89   260 10.65 758
```

In this case you specify all the numeric columns in the data frame, and the resultshowsthesumofeachofthegroupsrepresentedbythegroupingvariable (site).

You can also use the aggregate() command with a formula syntax; in this case you specify the response variable to the left of the ~ and the predictor variables to the right, like so:

```
> aggregate(Length ~ site, data = mf, FUN = mean) site Length
1         Exe   17.6
2         Lyn   22.0
3       Ouse    20.4
4         Taw   21.4
5Torridge       16.8
```

Thisallowsaslightlysimplercommandbecauseyoudonotneedthe$signsas long as you specify where the data are to be found. In this case you chose a single response variable (Length) and the mean() as your summary function. Youcanselectseveralresponsevariablesatoncebywrappingtheminacbind() command:

```
> aggregate(cbind(Length, BOD) ~ site, data = mf, FUN = mean)
      site Length   BOD
1      Exe   17.6 171.8
2      Lyn   22.0 106.8
```

3	Ouse	20.4	150.6

```
4          Taw      21.4149.0
5Torridge           16.8151.6
```

Here you chose two response variables (Length and BOD), which are givenin the cbind()command (thus making a temporary matrix). You can select all the variablesbyusingaperiodinsteadofanynamestotheleftofthe~:

```
> aggregate(. ~ site, data = mf, FUN = mean)
         site  Length  Speed  Algae  NO3   BOD
1         Exe    17.6   16.6     47  1.43 171.8
2         Lyn    22.0   14.6     71  2.59 106.8
3        Ouse    20.4   15.2     65  2.07 150.6
4         Taw    21.4   14.8     57  2.01 149.0
5    Torridge    16.8   17.8     52  2.13 151.6
```

Here you use all the variables in the data frame. This works only if the remaining variables are all numeric; if you have other character variables, you need to specify the columns you wantexplicitly.

Because of the nature of the output/result, some people may find the aggregate() command more useful in presenting summary statistics than the tapply() command discussed earlier. In the following example, you have a data frame with one response variable and three predictor variables:

```
> str(pw)
'data.frame': 18obs.of                4variables:
 $height:int            9 11 6 14 17 19 28 31 32 7...
 $ plant : Factor w/ 2 levels "sativa","vulgaris": 2 2 2 2 2 2 2 2
2 1 ...
 $ water : Factor w/ 3 levels "hi","lo","mid": 2 2 2 3 3 3 1 1 1 2
...
 $ season: Factor w/ 2 levels "spring","summer": 1 2 2 1 2 2 1 2 2
1 ...

> pw.agg = aggregate(height ~ plant * water * season, data = pw, FUN = mean)
            plant water season height
1          sativa    hi spring   44.0
2        vulgaris    hi spring   28.0
3          sativa    lo spring    7.0
4        vulgaris    lo spring    9.0
5          sativa   mid spring   14.0
6        vulgaris   mid spring   14.0
7          sativa    hi summer   37.5
8        vulgaris    hi summer   31.5
9          sativa    lo summer    5.5
10       vulgaris    lo summer    8.5
11         sativa   mid summer   16.0
```

12vulgaris midsummer 18.0

The result is a simple data frame, and this can make it easier to extract the components than for the array result you had previously with the tapply() command.

You could have achieved the same result by using the period instead of the grouping variable names like so:

>aggregate(height ~ . , data = pw, FUN = mean)

So, like the previous example, the period means, "everything else not already named." If you replace the period with a number 1, you get quite a different result:

>aggregate(height ~ 1 , data = pw, FUN = mean) height
1 19.44444

You get the overall mean value here; essentially you have said, "don't use any grouping variables."

The aggregate() command is very powerful, partly because you can use the formula syntax and partly because of the output, which is a single data frame. In the following activity you practice using the command using the mean as the summary function, but you could try some others (for example, median, sum, sd, orlength).

Summary

- Vectorobjectsneedtobeofequallengthbeforetheycanbemadeintoa dataframeormatrix.
- Youcanusetherep()commandtocreatereplicatelabelsasfactors.You canalsousethegl()commandtogeneratefactorlevels.
- The levels of a factor can be examined using the levels() and nlevels() commands.
- A character vector can be converted to a factor using the as.factor()and factor() commands.
- Data frames can be constructed using the data.frame() command. Matrix objects can be constructed using matrix(), cbind(), or rbind() commands. Simple summary commands can be applied to rows or columns using rowSums()and colMeanscommands.
- The apply()command can apply a function to rows or columns.
- The rowsum() command can use a grouping variable to sum data across rows. Grouping variables can be used in the tapply() and aggregate() commands along with any function.
- Ifmorethantwogroupingvariablesareusedwiththetapply()command,a multi-dimensional array object is the result. In contrast, the aggregate() commandalwaysproducesasingledataframeastheresultandcanusethe formulasyntax.

Exercises

1. Look at the `bees` data object from the `Beginning.RData` file for this exercise. The data are a matrix, and the columns relate to individual bee species and the rows to different flowers. The numerical data are the number of bees observed visiting each flower. Create a new factor variable that could be used as a grouping variable. Here you require the general color type to be represented; you can think of the first two as being equivalent (blue) and the last three as equivalent (yellow).

2. Take the `bees`matrix and add the grouping variable you just created to it toformanewmatrix.

3. Use the `flcol` grouping variable you just created to summarize the `Buff.tail` column in the `bees` data; use any sensible summarizing command.Canyouproduceasummaryforallthebeespeciesinonego?

4. Look at the `ChickWeight` data item, which comes built into R. The data comprise a data frame (although it also has other attributes) with a single response variable and some predictor variables. Look at median values for `weight` broken down by `Diet`. Now add the `Time` variable as a second groupingfactor.

5. Access the `mtcars` data, which are built in to R. The data are in a data frame with several columns. Summarize the miles-per-gallon variable (`mpg`) as a mean for the three grouping variables `cyl`, `gear`, and`carb`.

UNIT-V

Terminology

Topic	Key Points
Copy and Paste	You can copy and paste text from another application into R.Thisenablesyoutocreatehelpfilesandsnippetsof
Customized functions: function(args) expr	Create customized functions using the function() command;args=argumentstopasstoexpr(theactual function).Argsmayprovidedefaultvalues.
Multiple lines of text{ variouscommands }function(args) {various commands }	Curlybracketscanbeusedtoseparatesubroutines.This allowsmultiplelinestobeenteredintoconsole.
Annotations:# comment	Anythingfollowingthe#isignoredandsoitcanbeused forcomments.
Function arguments/instructions: args(function_name)	Theargs()commandreturnsthearguments/instructions requiredbythenamedfunction.
Looking at function code: function_name	Supplyingthefunctionnamewithout()andinstructions displaysthetextofthescript.
Read text files as scripts: source('filename')source(file.choose())	Readsatextfileandexecutesthelinesoftextas R commands.
Saving to disk: save(object, 'filename')	Savesabinaryversionofanobject(includingafunction) todisk.
Loading from disk: load('filename')	Loads a binary object from disk.
Save objects as text: dump('names_list', file = 'filename')	Attempts to write a text version of an object to disk.
Text messages on screen: cat('text1', 'text2')cat(chr_object)"\n""\'"	Produces a message in the console; requires plain text strings,explicitlyorfromcharacterobjects.Itemsmaybe separatedbycommas.Anewlineisproducedusing"\
Displaying results: print(object)	Printsthenamedobjecttotheconsole(thatis,thescreen).
Wait for user input: readline(prompt = "text")	Pausesandwaitsforinputfromtheuser.Amessagecan bedisplayedusingtheprompt=instruction.

Chapter 10

Regression (Linear Modeling)

What You Will Learn In This Chapter:

- How to carry out linear regression (including multiple regression)
- How to carry out curvilinear regression using logarithmic and polynomials asexamples
- How to build a regression model using both forward and backward stepwise processes
- How to plot regression models
- How to add lines of best-fit to regression plots
- How to determine confidence intervals for regression models How to plot
- confidence intervals
- How to draw diagnostic plots

Linear modeling is a widely used analytical method. In a general sense, it involves a response variable and one or more predictor variables. The technique uses a mathematical relationship between the response and predictor variables. You might, for example, have data on the abundance of an organism (the response variable) and details about various habitat variables (predictor variables). Linear modeling, or multiple regression as it is also known, can show you which of the habitat variables are most important, and also which are statistically significant. Linear regression is quite similar to the analysis of variance (ANOVA) that you learned about earlier. The main difference is that in ANOVA, the predictor variables are discrete (that is, they have different levels), whereas in regression they are continuous.

Simple Linear Regression

The simplest form of regression is akin to a correlation where you have two variables—a response variable and a predictor. In the following example you see asimpledataframewithtwocolumns,whichyoucancorrelate:

```
> 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


> cor.test(~ count + speed, data = fw)

        Pearson's product-moment correlation data:

        count andspeed
t = 2.5689, df = 6, p-value = 0.0424
alternative hypothesis: true correlation is not equal to 0
```

95 percent confidence interval: 0.03887166
0.94596455
sample estimates:
cor
0.7237206

Notethatinthissimplecorrelationyoudonothavearesponsetermtotheleft ofthe~intheformula.Youcanrunthesameanalysisusingthelm()command; thistime,though,youplacethepredictorontheleftofthe~andtheresponseon theright:

```
> lm(count ~ speed, data = fw)

Call:
lm(formula = count ~ speed, data = fw)

Coefficients: (Intercept)     speed
         8.2546               0.7914
```

The result shows you the coefficients for the regression, that is, the intercept and the slope. To see more details you should save your regression as a named object; then you can use the summary()command like so:

```
> fw.lm = lm(count ~ speed, data = fw)
> summary(fw.lm)

Call:
lm(formula = count ~ speed, data = fw)

Residuals:
      Min          1Q    Median          3Q         Max
-13.377       -5.801       -1.542        5.051      14.371

Coefficients:
                    Estimate Std. Error t value Pr(>|t|) (Intercept)
                       8.2546          5.8531        1.410        0.2081
speed                  0.7914          0.3081        2.569        0.0424*
---
Signif.codes:              0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’1

Residual standard error: 10.16 on 6 degrees of freedom MultipleR-squared:0.5238,
                                                    Adjusted R-squared:0.4444
F−statistic: 6.599 on 1 and6DF,                     p-value:0.0424
```

Now you see a more detailed result; for example, the p-value for the linear model is exactly the same as for the standard Pearson correlation that you ran earlier. The result object contains more information, and you can see what is available by using the names() command like so:

```
> names(fw.lm)
  [1]  "coefficients"          "residuals"          "effects"          "rank"
```

```
[5]  "fitted.values"    "assign"    "qr"          "df.residual"
[9]  "xlevels"          "call"      "terms"       "model"
```

You can extract these components using the $ syntax; for example, to get the coefficients you use the following:

```
> fw.lm$coefficients (Intercept)
                          speed
   8.2545956         0.7913603

> fw.lm$coef (Intercept)  speed
   8.2545956         0.7913603
```

In the first case you type the name in full, but in the second case you see that you can abbreviate the latter part as long as it is unambiguous.

Many of these results objects can be extracted using specific commands, as you see next.

Linear Model Results Objects

When you have a result from a linear model, you end up with an object that contains a variety of results; the basic summary() command shows you some of these. You can extract the components using the $ syntax, but some of these components are important enough to have specific commands. The following sections discuss these components and their commands in detail.

Coefficients

You can extract the coefficients using the coef() command. To use the command, you simply give the name of the linear modeling result like so:

```
> coef(fw.lm) (Intercept) speed
   8.2545956         0.7913603
```

You can obtain confidence intervals on these coefficients using the confint() command. The default settings produce 95-percent confidence intervals; that is, at 2.5 percent and 97.5 percent, likeso:

```
> confint(fw.lm)
                              2.5%          97.5%
(Intercept) -6.06752547 22.576717
speed                0.03756445       1.545156
```

You can alter the interval using the level = instruction, specifying the interval as a proportion. You can also choose which confidence variables to display (the default is all of them) by using the parm = instruction and placing the names of the variables in quotes as done in the followingexample:

```
> confint(fw.lm, parm = c('(Intercept)', 'speed'), level = 0.9)
                               5 %           95%
(Intercept) -3.1191134 19.628305
speed                0.1927440       1.389977
```

Note that the intercept term is given with surrounding parentheses like so,

(Intercept), which is exactly as it appears in the summary()command.

Fitted Values

You can use the fitted() command to extract values fitted to the linear model; in other words, you can use the equation of the model to predict y values for each xvalue like so:

```
> fitted(fw.lm)
      Taw  Torridge      Ouse       Exe       Lyn     Brook
Ditch       Fal
 9.837316 10.628676 12.211397 15.376838 19.333640 27.247243
31.204044 35.160846
```

In this case the rows of data are named, so the result of the fitted() command also produces names.

Residuals

You can view the residuals using the residuals() command; the resid() command is an alias for the same thing and produces the same result:

```
> residuals(fw.lm)
        Taw       Torridge          Ouse          Exe          Lyn
Brook
 -0.8373162  14.3713235   2.7886029-13.3768382  -5.3336397
-2.2472426
      Ditch         Fal
 -7.2040441  11.8391544
```

Onceagain,youseethattheresidualsarenamedbecausetheoriginaldatahad rownames.

Formula

You can access the formula used in the linear model using the formula() command likeso:

```
> formula(fw.lm)
count ~ speed
```

This is not quite the same as the complete call to the lm() command which looks likethis:

```
> fw.lm$call
lm(formula = count ~ speed, data = fw)
```

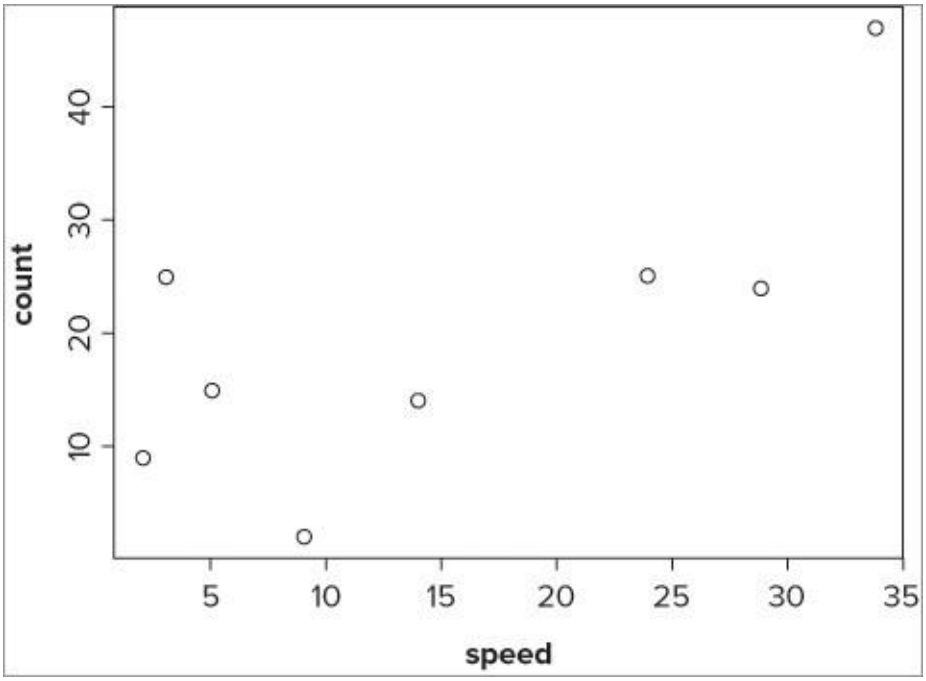
Best-Fit Line

You can use these linear modeling commands to help you visualize a simple linear model in graphical form. The following commands all produce essentially the samegraph:

```
> plot(fw$speed, fw$count)
> plot(~ speed + count, data = fw)
> plot(count ~ speed, data = fw)
> plot(formula(fw), data = fw)
```

The graph looks like Figure 10-1.

Figure 10-1:

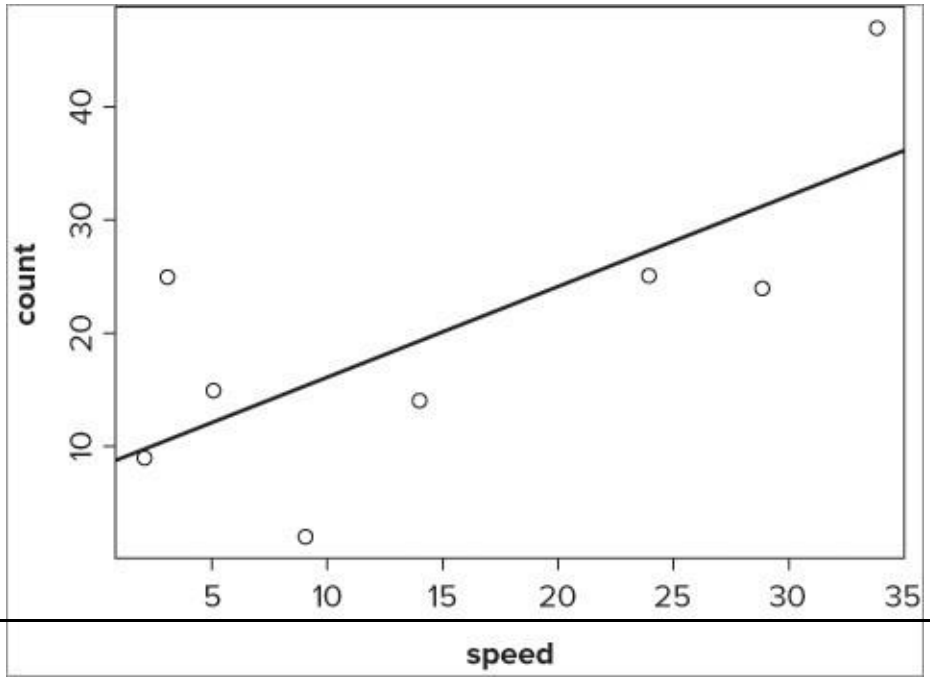


To add a line of best-fit, you need the intercept and the slope. You can use the abline() command to add the line once you have these values. Any of the following commands would produce the required line of best-fit:

```
> abline(lm(count ~ speed, data = fw))
> abline(a = coef(fw.lm[1], b = coef(fw.lm[2])))
> abline(coef(fw.lm))
```

The first is intuitive in that you can see the call to the linear model clearly. The second is quite clumsy, but shows where the values come from. The last is the most simple to type and makes best use of the lm() result object. The basic plot with a line of best-fit looks like Figure 10-2.

Figure 10-2:



You can draw your best-fit line in different styles, widths, and colors using options you met previously (lty, lwd, and col). Table 10-1acts as a reminder and summary of theiruse.

Table 10-1: Summary of Commands used in Drawing Lines of Best-Fit

Comman	Explanation
lty = n	Setsthelinetype.Linetypescanbespecifiedasaninteger(0=blank,1=solid(default),2=dashed,3=dotted,4=dotdash,5=longdash,6=twodash)orasoneofthecharacterstrings "blank", "solid", "dashed", "dotted", "dotdash", "longdash", or "twodash", where "blank"usesinvisiblelines(thatis,doesnotdrawthem).
lw = n d	Setsthelinewidthusingannumericalvaluewhere1isstandardwidth,2isdoublewidth,and soon.Defaultsto1.
col = color	Sets the line color using a named color (in quotes) or an integer value. Defaults to “black”

You look at fitting curves to linear models in a later section.

Simple regression, that is, involving one response variable and one predictor variable, is an important stepping stone to the more complicated multiple regression that you will meet shortly (where you have one response variable but several predictor variables). To put into practice some of the skills, you can try out regression for yourself in the followingactivity.

Similarity between lm() andaov()

You can think of the aov()command as a special case of linear modeling, with thecommandbeinga“wrapper”forthelm()command.Indeedyoucanusethe lm()command to carry out analysis of variance. In the following example, you see how to use the aov() and lm() commands with the same formula on the samedata:

```
> str(pw)
'data.frame': 18obs.of  4variables:
 $height:int  9 11 6 14 17 19 28 31 32 7...
 $ plant : Factor w/ 2 levels "sativa","vulgaris": 2 2 2 2 2 2 2 2
2 1 ...
 $ water : Factor w/ 3 levels "hi","lo","mid": 2 2 2 3 3 3 1 1 1 2
...
 $ season: Factor w/ 2 levels "spring","summer": 1 2 2 1 2 2 1 2 2
1 ...

> pw.aov = aov(height ~ water, data = pw)
> pw.lm = lm(height ~ water, data = pw)
```

You can use the summary() command to get the result in a sensible layout like so:

```
> summary(pw.aov)
             Df  Sum Sq Mean SqFvalue  Pr(>F) water
              22403.111201.56  84.484 6.841e-09***
Residuals    15  213.33   14.22
---
Signif.codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’1
```

```
> summary(pw.lm)

Call:
lm(formula = height ~ water, data = pw)

Residuals:
      Min       1Q   Median       3Q      Max
-7.0000 -2.0000 -0.6667  1.9167  9.0000

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)   35.000      1.540  22.733 4.89e-13 ***
waterlo      -27.667      2.177 -12.707 1.97e-09 ***
watermid     -19.000      2.177  -8.726 2.91e-07 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 3.771 on 15 degrees of freedom
Multiple R-squared: 0.9185,  Adjusted R-squared: 0.9076
F-statistic: 84.48 on 2 and 15 DF,  p-value: 6.841e-09
```

In the first case you see the "classic" ANOVA table, but the second summary looks a bit different. You can make the result of the lm() command look more like the usual ANOVA table by using the anova()command like so:

```
> anova(pw.lm)
Analysis of Variance Table

Response: height
          Df  Sum Sq Mean Sq F value    Pr(>F)
water      2 2403.11 1201.56  84.484 6.841e-09 ***
Residuals 15  213.33   14.22
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

ANOVA is essentially a special form of linear regression and the aov() commandproducesaresultthatmirrorsthelookandfeeloftheclassicANOVA. For most purposes you will use the aov() command for ANOVA and the lm() commandforlinearmodeling.

Multiple Regression

In the previous examples you used a simple formula of the form response ~ predictor. You saw earlier in the section on the aov() command that you can specify much more complex models; this enables you to create complex linear models. The formulae that you can use are essentially the same as you met previously, as you will see shortly. In multiple regression you generally have one response variable and several predictor variables. The main point of the regression is to determine which of the predictor variables is statistically important (significant), and the

relative effects that these have on the response variable.

Formulae and Linear Models

When you looked at the aov() command to carry out analysis of variance, you saw how to use the formula syntax to describe your ANOVA model. You can do the same with the lm() command, but in this case you should note that the Error() instruction is not valid for the lm() command and will work only in conjunction with the aov()command.

The syntax in other respects is identical to that used for the aov() command, and you can see some examples in Table 10-2. Note that you can specify intercept terms in your models. You can do this in aov() models as well but it makes lesssense.

Table 10-2: Formula Syntax and Regression Modeling

Formula	Explanation
y ~ xy ~ 1 + x	Linear regression of yon x. Implicit intercept.Explicit intercept.
y ~ 0 + xy ~ -1 + xy ~ x -1	Linear regression of yon xthrough origin, that is, without intercept.
log(y) ~ x1 + x2	Multiple regression of transformed variable yon x1and x2with implicit
y ~ poly(x, 2)y ~ 1 + x + I(x^2)	Polynomialregressionofyonxofdegree2.Firstformusesorthogonal polynomials.Secondformusesexplicitpowers.
y ~ X + poly(x, 2)	MultipleregressionywithmodelmatrixconsistingofmatrixXaswellasorthog polynomial terms in xto degree 2.
y ~ A	One-way analysis of variance.
y ~ A + x	Singleclassificationanalysisofcovariancemodelofy,withclassesdetermine dbyA
y ~ A * By ~ A + B + A:B	Two-factornon-additiveanalysisofvarianceofyonfactorsAandB,thatis,with interactions.
y ~ B %in% Ay ~ A/B	Nested analysis of variance with Bnested in A.
y ~ (A + B + C)^2y ~ A * B * C	Three-factorexperimentwithmodelcontainingmaineffectsandtwo-factor interactionsonly.
y ~ A * xy ~ A/xy ~ A/(1 + x) -1	Separate linear regression models of y on x within levels of A, with different coding.Lastformproducesexplicitestimatesofasmanyinterceptsandslopes

You can see from this table that you are able to construct quite complex models using the formula syntax. The standard symbols –, +, * , /, and ^ have specific meanings in this syntax; if you want to use the symbols in their regular mathematical sense, you use the I() instruction to "insulate" the terms from their formula meaning. So, the following examples are quite different in meaning:

y ~ x1 + x2
y ~ I(x1 + x2)

Model Building

When you have several or many predictor variables, you usually want to create the most statistically significant model from the data. You have two main choices: **forward stepwise regression** and **backward deletion**.

- **Forwardstepwiseregression**:Startoffwiththesinglebestvariableand add more variables to build your model into a more complex form **Backward deletion**: Put all
- the variables in and reduce the model by removingvariablesuntilyouareleftwithonlysignificantterms.

You can use the add1() and drop1() commands to take either approach.

Adding Terms with Forward Stepwise Regression

When you have many variables, finding a starting point is a key step. One option is to look for the predictor variable with the largest correlation with the response variable. You can use the cor() command to carry out a simple correlation. In the following example you create a correlation matrix and, therefore, get to see all the pairwise correlations; you simply select the largest:

```
> cor(mf)
                Length      Speed      Algae        NO3        BOD
Length       1.0000000 -0.34322968  0.7650757  0.45476093 -0.8055507
Speed       -0.3432297  1.00000000 -0.1134416  0.02257931  0.1983412
Algae        0.7650757 -0.11344163  1.0000000  0.37706463 -0.8365705
NO3          0.4547609  0.02257931  0.3770646  1.00000000 -0.3751308
BOD         -0.8055507  0.19834122 -0.8365705 -0.37513077  1.0000000
```

The response variable is Length in this example, but the cor() command has shown you all the possible correlations. You can see fairly easily that the correlation between Length and BOD is the best place to begin. You could begin your regression model likeso:

```
> mf.lm = lm(Length ~ BOD, data = mf)
```

In this example you have only four predictor variables, so the matrix is not too large; but if you have more variables, the matrix would become quite large and hard to read. In the following example you have a data frame with a lot more predictorvariables:

```
> names(pb)
 [1]"count"              "sward.may"       "mv.may"          "dv.may"
"sphag.may"       "bare.may"
 [7]"grass.may"  "nectar.may""sward.jul"                     "mv.jul"
"brmbl.jul"       "sphag.jul"
[13]"bare.jul"          "grass.jul"       "nectar.jul""sward.sep"           "mv.sep"
"brmbl.sep"
[19]"sphag.sep"         "bare.sep"        "grass.sep"       "nectar.sep"

> cor(pb$count, pb)
      countsward.may          mv.may       dv.maysphag.may          bare.may
grass.may
[1,]          1 0.3173114 0.386234 0.06245646 0.4609559-0.3380889
-0.2345140
      nectar.maysward.jul               mv.jul      brmbl.julsphag.jul bare.jul
```

```
                grass.jul
[1,]       0.781714 0.1899664 0.1656897 -0.20907260.2877822
-0.2283124 -0.1625899
      nectar.julsward.sep                mv.sep      brmbl.sepsphag.sep                bare.sep
grass.sep
[1,]       0.259654 0.6476513 0.877378 -0.2098358 0.7011718-0.4196179
-0.6777093
      nectar.sep
[1,]       0.7400115
```

If you had used the plain form of the cor() command, you would have a lot of searching to do, but here you limit your results to correlations between the response variable and the rest of the data frame. You can see here that the largest correlation is between count and mv.sep, and this would make the best starting point for the regression model:

```
> pb.lm = lm(count ~ mv.sep, data = pb)
```

It so happens that you can start from an even simpler model by including no predictor variables at all, but simply an explicit intercept. You replace the name of the predictor variable with the number 1 likeso:

```
> mf.lm = lm(Length ~ 1, data = mf)
> pb.lm = lm(count ~ 1, data = pb)
```

In both cases you produce a "blank" model that contains only an intercept term. You can now use the add1() command to see which of the predictor variables is the best one to add next. The basic form of the command is as follows:

```
add1(object, scope)
```

The object is the linear model you are building, and the scope is the data that form the candidates for inclusion in your new model. The result (shown here) is alistoftermsandthe"effect"theywouldhaveifaddedtoyourmodel:

```
> add1(mf.lm, scope = mf) Single
term additions

Model:
Length ~ 1
```

	Df	Sum of Sq	RSS	AIC
<none>			227.760	57.235
Speed	1	26.832	200.928	56.102
Algae	1	133.317	94.443	37.228
NO3	1	47.102	180.658	53.443
BOD	1	147.796	79.964	33.067

In this case you are primarily interested in the AIC column. You should look to add the variable with the lowest AIC value to the model; in this instance you see that BOD has the lowest AIC and so you should add that. This ties in with the correlation that you ran earlier. The new model then becomes:

```
> mf.lm = lm(Length ~ BOD, data = mf)
```

You can now run the add1()command again and repeat the process like so:

```
> add1(mf.lm, scope = mf) Single
term additions

Model: Length
~BOD
       Df  Sum of Sq    RSS    AIC
<none>                 79.964 33.067
Speed   1    7.9794  71.984 32.439
Algae   1    6.3081  73.656 33.013
NO3     1    6.1703  73.794 33.060
```

You can see now that Speed is the variable with the lowest AIC, so this is the next variable to include. Note that terms that appear in the model are not included in the list. If you now add the new term to the model, you get the followingresult:

```
> mf.lm = lm(Length ~ BOD + Speed, data = mf)
> summary(mf.lm)

Call:
lm(formula = Length ~ BOD + Speed, data = mf)

Residuals:
     Min       1Q   Median       3Q      Max
-3.1700 -0.5450 -0.1598   0.8095   2.9245

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept) 29.30393    1.62068  18.081 1.08e-14***
BOD         -0.05261    0.00838  -6.278 2.56e-06***
Speed       -0.12566    0.08047  -1.562    0.133
---
Signif.codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’1

Residual standard error: 1.809 on 22 degrees of freedom MultipleR-squared:0.6839,
                                                    Adjusted R-squared:0.6552
F−statistic:  23.8 on 2 and22DF,  p-value:3.143e-06
```

You can see that the Speed variable is not a statistically significant one and probably should not to be included in the final model. It would be useful to see the level of significance before you include the new term. You can use an extra instruction in the add1() command to do this; you can use test = ‘F’ to show the significance of each variable if it were added to your model. Note that the ‘F’ is not short for FALSE, but is for an F-test. If you run the add1() command again, you see something likethis:

```
> mf.lm = lm(Length ~ BOD, data = mf)

> add1(mf.lm, scope = mf, test = 'F')
Single term additions
```

Model: Length
~BOD

	Df	Sum of Sq	RSS	AIC	F value	Pr(F)
<none>			79.964	33.067		
Speed	1	7.9794	71.984	32.439	2.4387	0.1326
Algae	1	6.3081	73.656	33.013	1.8841	0.1837
NO3	1	6.1703	73.794	33.060	1.8395	0.1888

Now you can see that none of the variables in the list would give statistical significance if added to the current regression model.

In this example the model was quite simple, but the process is the same regardless of how many predictor variables arepresent.

Removing Terms with BackwardsDeletion

You can choose a different approach by creating a regression model containing all the predictor variables you have and then trim away the terms that are not statistically significant. In other words, you start with a big model and trim it down until you get to the best (most statistically significant). To do this you can use the drop1() command; this examines a linear model and determines the effect of removing each one from the existing model. Complete the following steps to perform a backwards deletion.

1. Tostart,createa"full"model.Youcouldtypeinallthevariablesatonce, butthiswouldbesomewhattedioussoyoucanuseashortcut:

```
> mf.lm = lm(Length ~ ., data = mf)
```

2. In this instance you use Lengthas the response variable, but on the right of the ~ you use a period to represent "everything else." Check what the actualformulahasbecomeusingtheformula()command:

```
> formula(mf.lm)
Length ~ Speed + Algae + NO3 + BOD
```

3. Now use the drop1() command and see which of the terms you should delete:

```
> drop1(mf.lm, test = 'F')
Single term deletions

Model:
Length ~ Speed + Algae + NO3 + BOD
       Df SumofSq      RSS     AICFvalue      Pr(F)
<none                57.912 31.001
Speed   1  10.9550  68.867 33.333  3.7833  0.06596 .
Algae   1   6.2236  64.136 31.553  2.1493  0.15818
NO3     1   6.2261  64.138 31.554  2.1502  0.15810
BOD     1  12.3960  70.308 33.850  4.2810  0.05171 .
---
Signif.codes:      0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

4. Look to remove the term with the lowest AIC value; in this case the Algae variable has the lowest AIC. Re-form the model without this variable. The simplest way to do this is to copy the model formula to the clipboard, paste it into a new command, and edit out the term you do not want:

```
> mf.lm = lm(Length ~ Speed + NO3 + BOD, data =mf)
```

5. Examinetheeffectofdroppinganothertermbyrunningthedrop1() command oncemore:

```
> drop1(mf.lm, test = 'F')
Single term deletions

Model:
Length ~ Speed + NO3 + BOD
       Df SumofSq     RSS    AICFvalue      Pr(F)
<none                64.136  31.55
Speed   1    9.658  73.794  33.06  3.1622   0.08984  .
NO3     1    7.849  71.984  32.43  2.5699   0.12385
BOD     1   88.046 152.18  51.15  28.829 2.520e-05  **
---
Signif.codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’1
```

You can see now that the NO3 variable has the lowest AIC and can be removed. You can carry out this process repeatedly until you have a model that you are happywith.

Comparing Models

It is often useful to compare models that are built from the same data set. This allows you to see if there is a statistically significant difference between a complicated model and a simpler one, for example. This is useful because you always try to create a model that most adequately describes the data with the minimum number of terms. You can compare two linear models using the anova() command. You used this earlier to present the result of the lm() command as a classic ANOVA table likeso:

```
> mf.lm = lm(Length ~ BOD, data = mf)

> anova(mf.lm)
Analysis of Variance Table

Response: Length
           Df  Sum Sq Mean SqFvalue          Pr(>F)  BOD
            1147.796147.796       42.511 1.185e-06***
Residuals23    79.964   3.477
---
Signif.codes:       0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’1
```

You can also use the anova()command to compare two linear models (based on the same data set) by specifying both in the command like so:

```
> mf.lm1 = lm(Length ~ BOD, data = mf)
> mf.lm2 = lm(Length ~ ., data = mf)

> anova(mf.lm1, mf.lm2) Analysis of
Variance Table

Model 1: Length ~ BOD
Model 2: Length ~ Speed + Algae + NO3 + BOD Res.Df
            RSS Df SumofSq          F     Pr(>F)
```

```
1         2379.964
2         2057.912        3        22.052 2.5385 0.08555.
---
Signif.codes:              0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’1
```

In this case you create two models; the first contains only a single term (the most statistically significant one) and the second model contains all the terms. The anova() command shows you that there is no statistically significant difference between them; in other words, it is not worth adding anything to the original model!

You do not need to restrict yourself to a comparison of two models; you can include more in the anova()command like so:

```
> anova(mf.lm1, mf.lm2, mf.lm3) Analysis
of Variance Table

Model 1: Length ~ BOD
Model 2: Length ~ BOD + Speed  Model 3:
Length ~ BOD + Speed +NO3
   Res.Df          RSS Df SumofSq                  FPr(>F) 1
          2379.964
2         2271.984        1        7.9794 2.61270.1209
3         2164.136        1        7.8486 2.56990.1239
```

Here you see a comparison of three models. The conclusion is that the first is the minimum adequate model and the other two do not improvematters.

Building the best regression model is a common task, and it is a useful skill to employ; in the following activity you practice by creating a regression model with the forward stepwiseprocess.

Curvilinear Regression

Your linear regression models do not have to be in the form of a straight line; as long as you can describe the mathematical relationships, you can carry out linear regression. When your mathematical relationship is not in straight-line form then it is described as curvilinear). The basic relationship for a linear regression is:

$y = mx + c$

In this classic formula, y represents the response variable and x is the predictor variable; this relationship forms a straight line. The m and c terms represent the slope and intercept, respectively. When you have multiple regression, you simply add more predictor variables and slopes, like so:

$y = m_1x_1 + m_2x_2 + m_3x_3 + m_nx_n + c$

The equation still has the same general form and you are dealing with straight lines. However, the world is not always working in straight lines and other mathematical relationships are probable. In the following example you see two cases by way of illustration; the first case is a logarithmic relationship and the

second is a polynomial.

In the logarithmic case the relationship can be described as follows: $y = m \log(x) + c$

In the polynomial case the relationship is:

$$y = m_1x + m_2x^2 + m_3x^3 + m_nx^n + c$$

The logarithmic example is more akin to a simple regression, whereas the polynomial example is a multiple regression. Dealing with these non-straight regressions involves a slight deviation from the methods you have already seen.

Logarithmic Regression

Logarithmic relationships are common in the natural world; you may encounter them in many circumstances. Drawing the relationships between response and predictor variables as scatter plots is generally a good starting point. This can help you to determine the best approach to take.

The following example shows some data that are related in a curvilinear fashion:

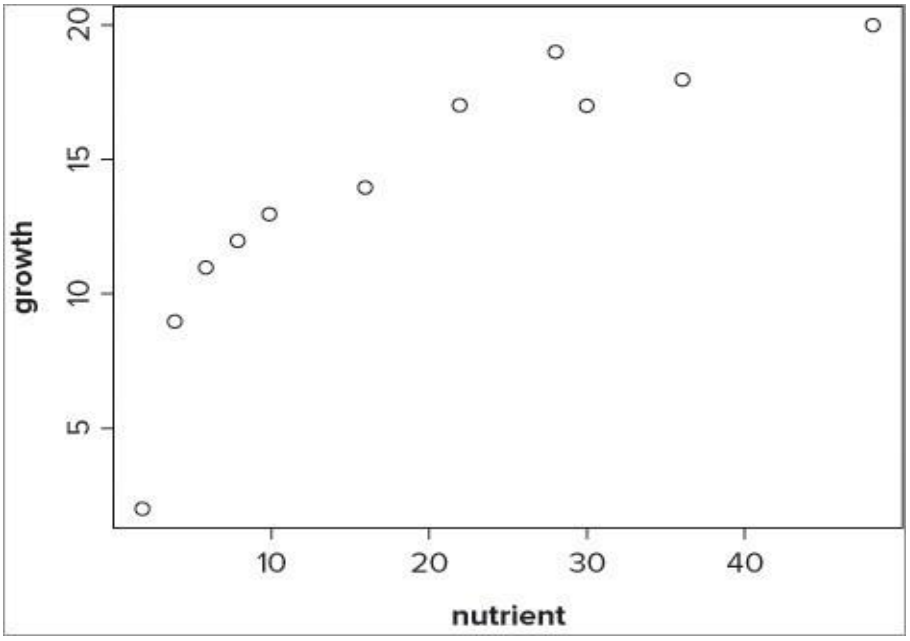
```
> pg
   growth nutrient
1       2        2
2       9        4
3      11        6
4      12        8
5      13       10
6      14       16
7      17       22
8      19       28
9      17       30
10     18       36
11     20       48
```

Here you have a simple data frame containing two variables: the first is the response variable and the second is the predictor. You can see the relationship more clearly if you plot the data as a scatter plot using the plot() command like so:

```
> plot(growth ~ nutrient, data = pg)
```

This produces a basic scatter graph that looks like Figure 10-3.

Figure 10-3:



You can see that the relationship appears to be a logarithmic one. You can carry out a linear regression using the log of the predictor variable rather than the basic variable itself by using the lm()command directly like so:

```
> pg.lm = lm(growth ~ log(nutrient), data = pg)
> summary(pg.lm)

Call:
lm(formula = growth ~ log(nutrient), data = pg)

Residuals:
      Min        1Q    Median        3Q       Max
-2.2274-0.9039         0.5400    0.9344    1.3097

Coefficients:
                    Estimate Std. Error t value Pr(>|t|)
(Intercept)          0.6914     1.0596    0.652     0.53
log(nutrient)        5.1014     0.3858   13.223 3.36e-07***
---
Signif.codes:        0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 1.229 on 9 degrees of freedom Multiple R-squared:
0.951, Adjusted R-squared: 0.9456
F-statistic: 174.8 on 1 and9DF,                p-value:3.356e-07
```

Here you specify that you want to use log(nutrient) in the formula; note that you do not need to use the I(log(nutrient)) instruction because log has no formula meaning.

You could now add the line of best-fit to the plot. You cannot do this using the abline()

command though because you do not have a straight line, but a curved one. You see how to add a curved line of best-fit shortly in the section "Plotting Linear Models and Curve Fitting"; before that, you look at a polynomial example.

Polynomial Regression

A polynomial regression involves one response variable and one predictor variable, but the predictor variable is encountered more than once. In the simplest polynomial, the equation can be written like so:

$$y = m_1x + m_2x^2 + c$$

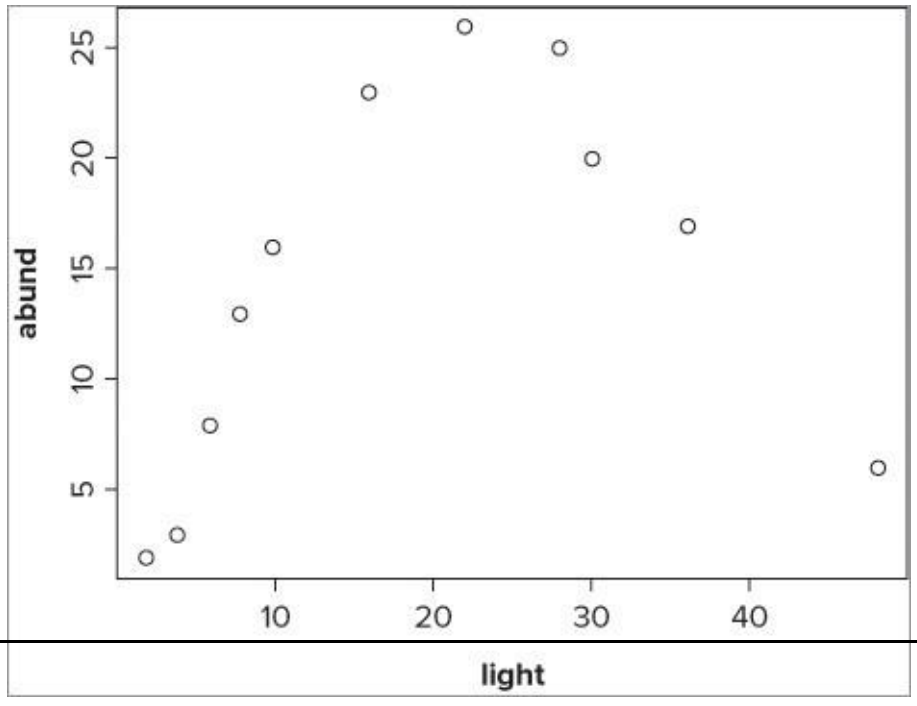
Polynomial relationships can occur in the natural world; examples typically include the abundance of a species in response to some environmental factor.The following example shows a typical situation. Here you have a data frame with two columns; the first is the response variable and the second is the predictor:

```
> bbel
   abund light
1      2     2
2      3     4
3      8     6
4     13     8
5     16    10
6     23    16
7     26    22
8     25    28
9     20    30
10    17    36
11     6    48
```

It appears that the response variable increases and then decreases again. If you plot the data (see Figure 10-4), you can see the situation more clearly:

```
> plot(abund ~ light, data = bbel)
```

Figure 10-4:



The relationship in Figure 10-4 looks suitable to fit a polynomial model—that is, $y = x + x^2 + c$—and you can therefore specify this in the lm()model like so:

```
> bbel.lm = lm(abund ~ light + I(light^2), data = bbel)
```

Notice now that you place the light^2 part inside the I() instruction; this ensures that the mathematical meaning is used. You can use the summary() command to see the final result:

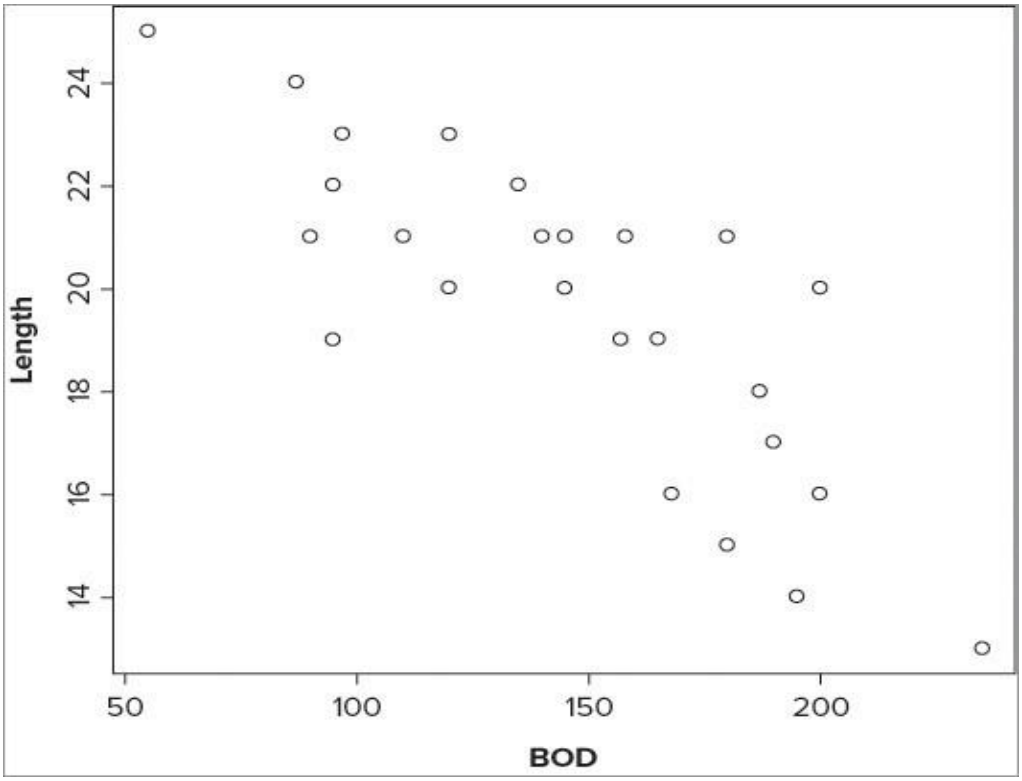
```
> summary(bbel.lm)

Call:
lm(formula = abund ~ light + I(light^2), data = bbel)

Residuals:
     Min       1QMedian           3Q      Max
-3.538-1.748        0.909      1.690    2.357

Coefficients:
                    Estimate      Std. Error     t value  Pr(>|t|)
(Intercept)       -2.004846       1.735268      -1.155       0.281
Light              2.060100       0.187506      10.987  4.19e-06 ***
I(light^2)        -0.040290       0.003893     -10.348  6.57e-06 ***
---
Signif.codes:          0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ''1 Residual standard

error: 2.422 on 8 degrees offreedom

MultipleR-squared:0.9382,                    Adjusted R-squared:0.9227
F-statistic: 60.68 on 2 and8DF,                 p-value:1.463e-05
```

The next step in the modeling process would probably be to add the line representing the fit of your model to the existing graph. This is the subject of the followingsection.

Plotting Linear Models and Curve Fitting

When you carry out a regression, you will naturally want to plot the results as some sort of graph. In fact, it would be good practice to plot the relationship between the variables before conducting the analysis. The pairs() command that you saw in Chapter 7 makes a good start, although if you have a lot of predictor variables, the individual plots can be quite small.

Figure 10-5:



If you have only one predictor variable, you can also add a line of best-fit, that is, one that matches the equation of the linearmodel.

Best-Fit Lines

If you want to add a best-fit line, you have two main ways to do this: The

- abline()command produces straight lines.
- The lines()command can produce straight or curved lines.

You have already met the abline() command, which can only draw straight lines, so this will only be mentioned briefly. To create the best-fit line you need to determine the coordinates to draw; you will see how to calculate the appropriatecoordinatesusingthefitted()command.Thelines()commandis able to draw straight or curved lines, and later you will see how to use the spline()commandtosmoothoutcurvedlinesofbest-fit.

Adding Line of Best-Fit with abline()

To add a straight line of best-fit, you need to use the abline() command. This requires the intercept and slope, but the command can get them directly from the result of an lm()command. So, you could create a best-fit line like so:

```
> abline(mf.lm)
```

This does the job. You can modify the line by altering its type, width, color, and so on using commands you have seen before (lty, lwd, and col).

Calculating Lines with fitted()

You can use the fitted() command to extract the fitted values from the linear model. These fitted values are determined from the x values (that is, the predictor) using the equation of the linear model. You can add lines to an existing plot using the lines() command; in this case you can use the original x values and the fitted yvalues to create your line of best-fit like so:

```
> lines(mf$BOD, fitted(mf.lm))
```

The final graph looks like Figure 10-6.

You can do something similar even when you have a curvilinear fit line. Earlier you created two linear models with curvilinear fits; the first was a logarithmic model and the second was a polynomial:

```
> pg.lm = lm(growth ~ log(nutrient), data = pg)
> bbel.lm = lm(abund ~ light + I(light^2), data = bbel)
```

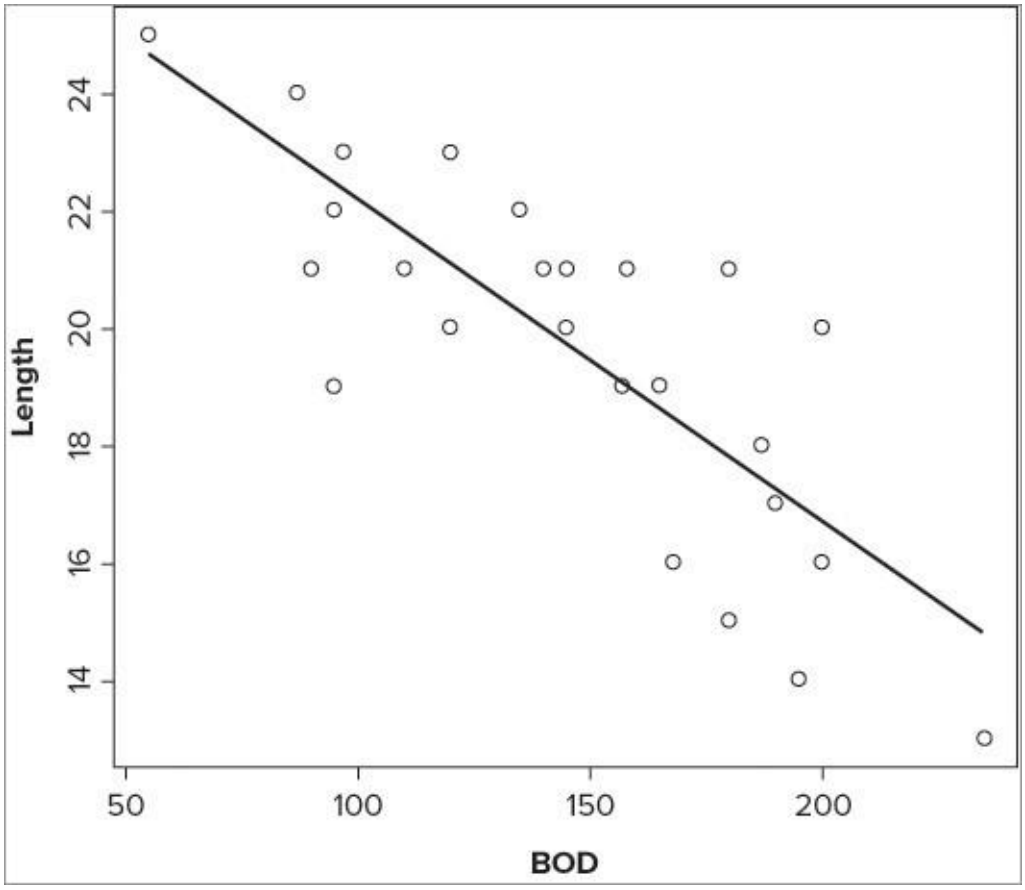
When you come to plot these relationships, you start by plotting the response against the predictor, as shown in the following two examples:

```
> plot(growth ~ nutrient, data = pg)
> plot(abund ~ light, data = bbel)
```

The first example plots the logarithmic model and the second plots the polynomial model. You can now add the fitted curve to these plots in exactly the same way as you did to produce Figure 10-6 by using the original x values and the fitted values from the linearmodel:

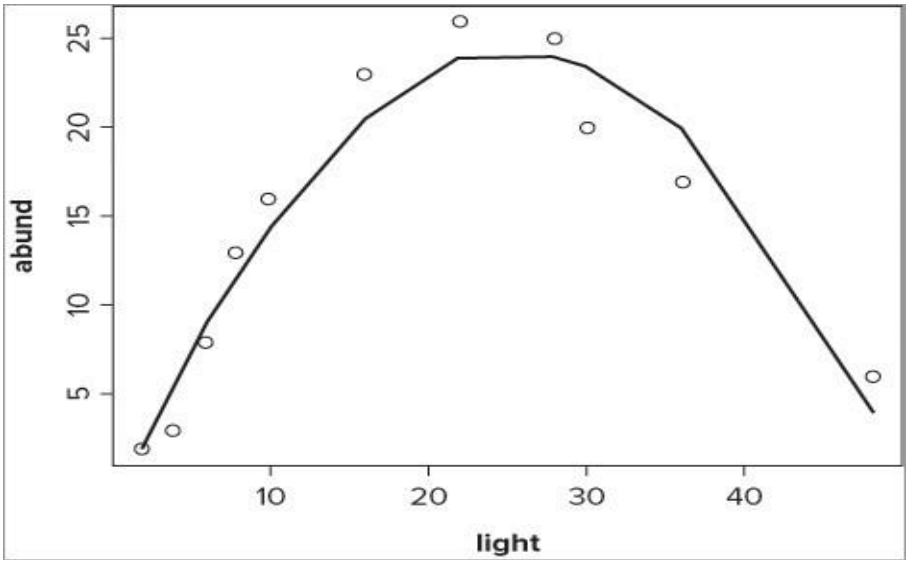
```
> lines(pg$nutrient, fitted(pg.lm))
> lines(bbel$light, fitted(bbel.lm))
```

Figure 10-6



This produces a line of best-fit, although if you look at the polynomial graph as an example (Figure 10-7), you see that the curve is really a series of straight lines.

Figure 10-7



What you really need is a way to smooth out the sections to produce a shapelier curve. You can do this using the spline()command, as you see next.

Producing Smooth Curves using spline()

You can use spline curve smoothing to make your curved best-fit lines look better; the spline() command achieves this. Essentially, the spline() commandrequiresaseriesofx,ycoordinatesthatmakeupacurve.Becauseyou already have your curve formed from the x values and fitted y values, you simplyenclosethecoordinatesofyourline()inaspline()commandlikeso:

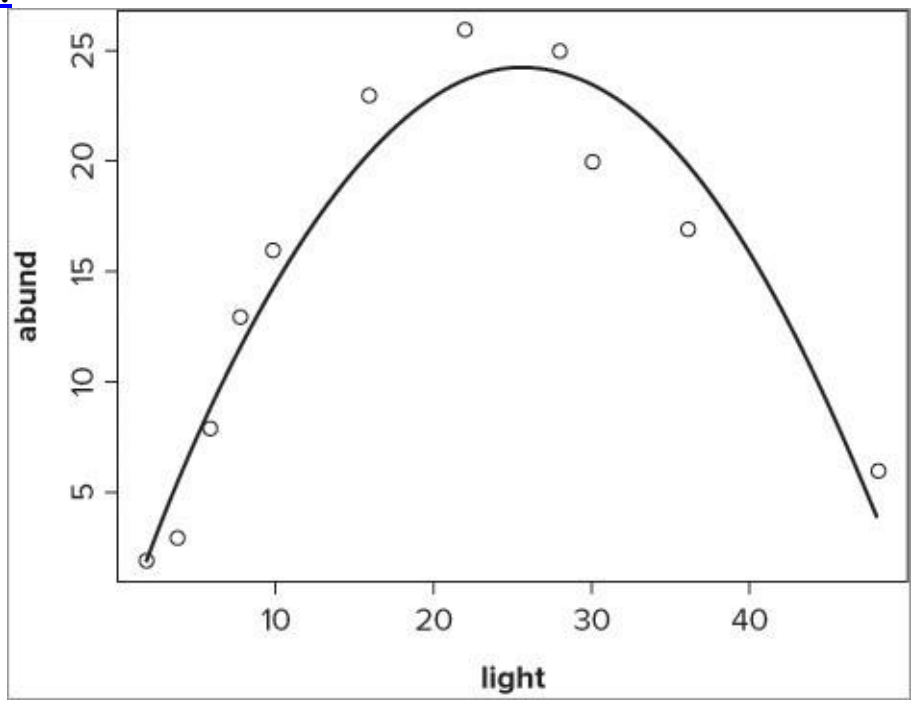
```
> lines(spline(bbel$light, fitted(bbel.lm)))
```

Thisaddsacurvedfitlinetothepolynomialregressionyoucarriedoutearlier.
Thecompletesetofinstructionstoproducethepolynomialmodelandgraphare shownhere:

```
> bbel.lm = lm(abund ~ light + I(light^2), data = bbel)
> plot(abund ~ light, data = bbel)
> lines(spline(bbel$light, fitted(bbel.lm)), lwd = 2)
```

The final graph appears like Figure 10-8.

Figure 10-8:



4. ddthelineofbest-fit:
   > lines(spline(pg$nutrient, fitted(pg.lm)), lwd =1.5)

5. Redrawthegraphusingthelogarithmofthepredictorvariable:
   > plot(growth ~ log(nutrient), data = pg, ylab = 'Plant growth', xlab = 'log(Nutrient)')

6. Nowaddalineofbest-fit:
   > abline(coef(pg.lm))

How It Works

The regular lm() command carries out a simple regression where you specify the predictor variable in terms of its logarithm. When the basic graph is plotted, the curved nature of the relationship becomes clear. The line of best-fit is curved and the lines() command adds it to the existing plot. The spline() command curves the line and makes it smooth. The y values for the line are taken from the fitted values (via the fitted() command).

If you plot the graph using the same formula as the linear regression, you see that the points are more nearly in a straight line. In this case you can use the abline() command to draw the line of best-fit. This time you use thecoefficientsfromtheregressiontoformthecoordinatesfortheline.

Confidence Intervals on FittedLines

After you have drawn your fitted line to your regression model, you may want to add confidence intervals. You can use the predict() command to help you do this. If you run the predict() command using only the name of your linear model result, you get a list of values that are identical to the fitted values; in other words, the two following commands areidentical:

```
>fitted(mf.lm)
>predict(mf.lm)
        1         2         3         4         5         6         7
8         9
16.65687  17.76092  20.24502  21.07305  21.62507  21.07305  22.45310
18.42334  17.76092
       10        11        12        13        14        15        16
17        18
16.93288  18.97537  19.69299  19.96901  19.69299  18.58895  17.37450
17.20889  19.03057
       19        20        21        22        23        24        25
22.72912  14.72480  16.65687  24.66119  22.89472  22.34270  22.45310
```

However, you can make the command produce confidence intervals for each of the predicted values by adding the interval = “confidence” instruction like so:

```
> predict(mf.lm, interval = 'confidence')
       fit       lwr       upr
1 16.65687  15.43583  17.87791
2 17.76092  16.78595  18.73588
3 20.24502  19.45005  21.03998
4 21.07305  20.17759  21.96851
5 21.62507  20.62918  22.62096
6 21.07305  20.17759  21.96851
```

7	22.45310	21.27338	23.63282
8	18.42334	17.56071	19.28597
9	17.76092	16.78595	18.73588
10	16.93288	15.77840	18.08737
11	18.97537	18.17562	19.77511
12	19.69299	18.92137	20.46462
13	19.96901	19.19054	20.74747
14	19.69299	18.92137	20.46462
15	18.58895	17.74852	19.42938
16	17.37450	16.32009	18.42891
17	17.20889	16.11799	18.29980
18	19.03057	18.23527	19.82587
19	22.72912	21.48183	23.97640
20	14.72480	12.98494	16.46466
21	16.65687	15.43583	17.87791
22	24.66119	22.89113	26.43126
23	22.89472	21.60574	24.18371
24	22.34270	21.18925	23.49615
25	22.45310	21.27338	23.63282

The result is a matrix with three columns; the first shows the fitted values, the second shows the lower confidence level, and the third shows the upper confidence level. By default, level = 0.95 (that is both upper and lower confidence levels are set to 95 percent), but you can alter this by changing the value in the level =instruction.

Now that you have values for the confidence intervals and the fitted values, you have the data you need to construct the model line as well as draw the confidence bands around it. To do so, perform the following steps:

1. Begin by using the predict() command to produce the confidence intervals,whichyoumakeintoanamedobject:

```
>prd = predict(mf.lm, interval = 'confidence', level =0.95)
```

2. Next add your x data to the result object. You do it in this way because you want to make sure that the values are sorted in order of increasing x value. Your result object is a matrix, but convert it to a data frame as you add the xdata like so:

```
> attach(mf)
> prd = data.frame(prd, BOD)
> detach(mf)
```

Notice that you used the attach() and detach() commands to get the BOD data. You could have used mf$BOD, but then the column would be named mf.BOD. You could, of course, rename the columns in the new data frame, but in the end it is easier to keep the name of your x variable as it was originally written. You cannot use the with() command in this case; if you try it you get anerror:

```
> with(mf, prd = data.frame(prd,BOD))
Error in eval(expr, envir, enclos) : argument is missing, with nodefault
```

Instead, make the result matrix into a data frame first and then add the original xvalues (BOD) like so:

```
> prd = data.frame(prd)
```

```
> prd$BOD = mf$BOD
```

You now have a data frame containing the original x values, as well as the fitted values and lower and upper confidence intervals:

```
> head(prd)
       fit      lwr      upr BOD
1 16.65687 15.4358 17.8779 200
2 17.76092 16.7859 18.7358 180
3 20.24502 19.4500 21.0399 135
4 21.07305 20.1775 21.9685 120
5 21.62507 20.6291 22.6209 110
6 21.07305 20.1775 21.9685 120
```

3. The only remaining problem is that the x values (that is, BOD) are not in numerical order, and if you try to make a line it will not come out looking right;tocounteractthis,re-sortthedataframeusingtheascendingnumerical values of the y data. Use the order() command to alter the sort order of your dataframe:

```
> prd = prd[order(prd$BOD),]
> head(prd)
       fit      lwr      upr BOD
22 24.6611 22.8911 26.4312  55
23 22.8947 21.6057 24.1837  87
19 22.7291 21.4818 23.9764  90
7  22.4531 21.2733 23.6328  95
25 22.4531 21.2733 23.6328  95
24 22.3427 21.1892 23.4961  97
```

4. Now at last you have the data you need sorted in the correct order; you can build your plot starting with the originaldata:

```
> plot(Length ~ BOD, data = mf)
```

5. Nowaddyourlinesusingthedataframeyoucreated:

```
> lines(prd$BOD, prd$fit)
```

6. This makes the line of best-fit and gives the same result as using the abline() command. Follow up by adding the lower and upper confidence bands likeso:

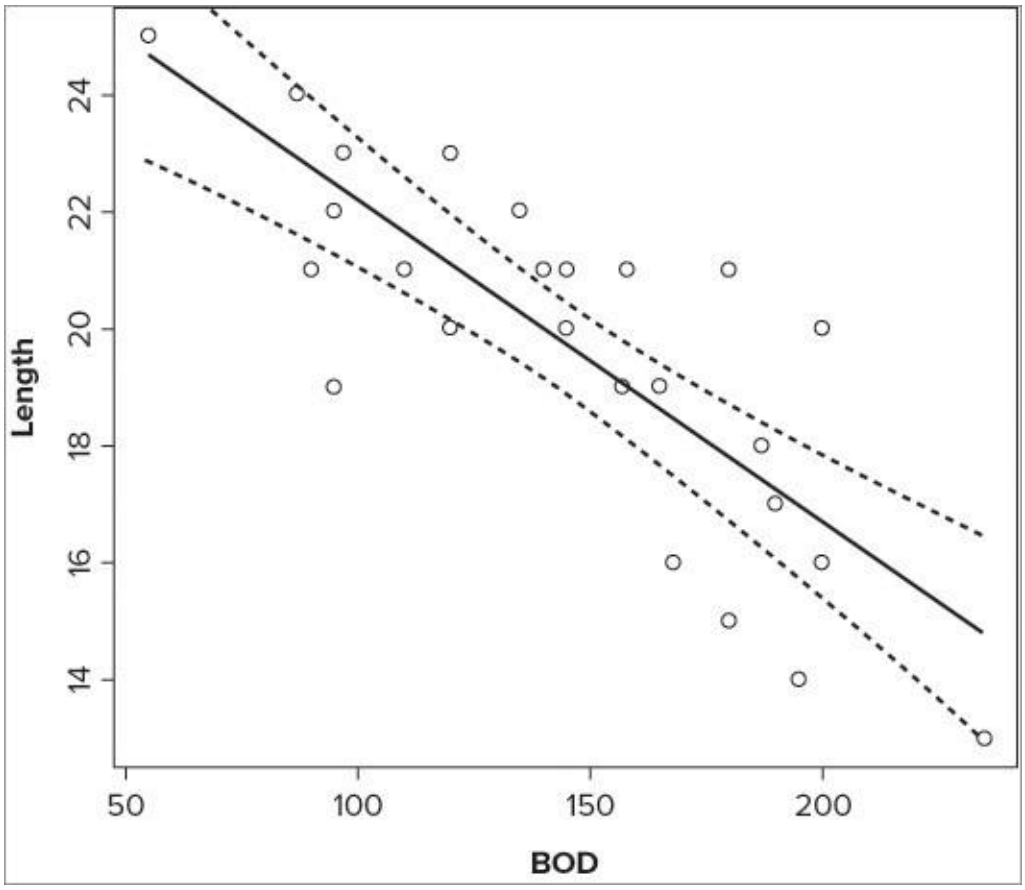
```
> lines(prd$BOD, prd$lwr, lty = 2)
> lines(prd$BOD, prd$upr, lty = 2)
```

7. In this case, make the confidence lines dashed using the lty = 2 instruction. The full process is shown in the following command lines:

```
> prd = predict(mf.lm, interval = 'confidence', level = 0.95) # makeCI
> attach(mf)
> prd = data.frame(prd, BOD) # addy-data
> detach(mf)
> prd = prd[order(prd$BOD),] # re-sort in order ofy-data
> plot(Length ~ BOD, data = mf) # basicplot
> lines(prd$BOD, prd$fit) # also bestfit
> lines(prd$BOD, prd$lwr, lty = 2) # lowerCI
> lines(prd$BOD, prd$upr, lty = 2) # upperCI
```

This produces a plot that looks like Figure 10-9.

Figure 10-9:



If you use these commands on a curvilinear regression, your lines will be a little angular so you can use the spline() command to smooth out the lines as you saw earlier:

```
>lines(spline(x.values, y.values))
```

If you were looking at the logarithmic regression you conducted earlier, for example, you would use the following:

```
> plot(growth ~ nutrient, data = pg)
> prd = predict(pg.lm, interval = 'confidence', level = 0.95)
> prd = data.frame(prd)
> prd$nutrient = pg$nutrient
> prd = prd[order(prd$nutrient),]
> lines(spline(prd$nutrient, prd$fit))
> lines(spline(prd$nutrient, prd$upr), lty = 2)
> lines(spline(prd$nutrient, prd$lwr), lty = 2)
```

Confidence intervals are an important addition to a regression plot. To practice the skills required to use them you can carry out the following activity.

Summarizing RegressionModels

Once you have created your regression model you will need to summarize it. This will help remind you what you have done, as well as being the foundation for presenting your results to others. The simplest way to summarize your regression model is using the summary() command,

as you have seen before. Graphical summaries are also useful and can often help readers visualize the situation more clearly than the numerical summary.

Diagnostic Plots

You can produce several diagnostic plots quite simply by using the plot() command like so:

plot(my.lm)

Once you type the command, R opens a blank graphics window and gives you a message. This message should be self-explanatory:

Hit <Return> to see next plot:

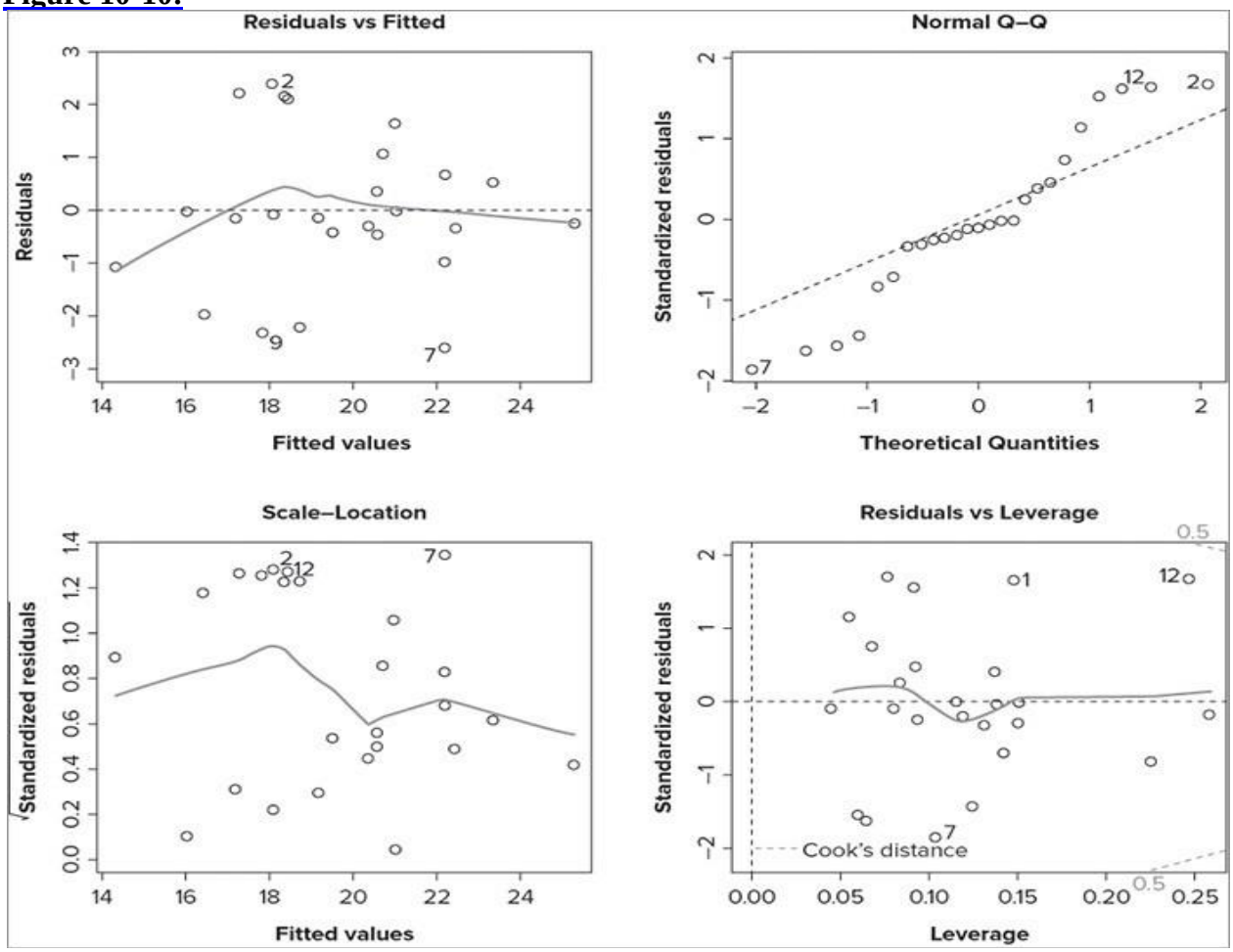
Each time you press the Enter key, a plot is produced; you have four in total and you can stop at any time by pressing the ESC key. The first plot shows residuals against fitted values, the second plot shows a normal QQ plot, the third shows (square root) standardized residuals against fitted values, and the fourth shows the standardized residuals against the leverage. In the following example, youcanseeasimpleregressionmodelandthestartofthediagnosticprocess:

> mf.lm = lm(Length ~ BOD + Speed, data = mf)

> plot(mf.lm)

Hit <Return> to see next plot:

Figure 10-10:



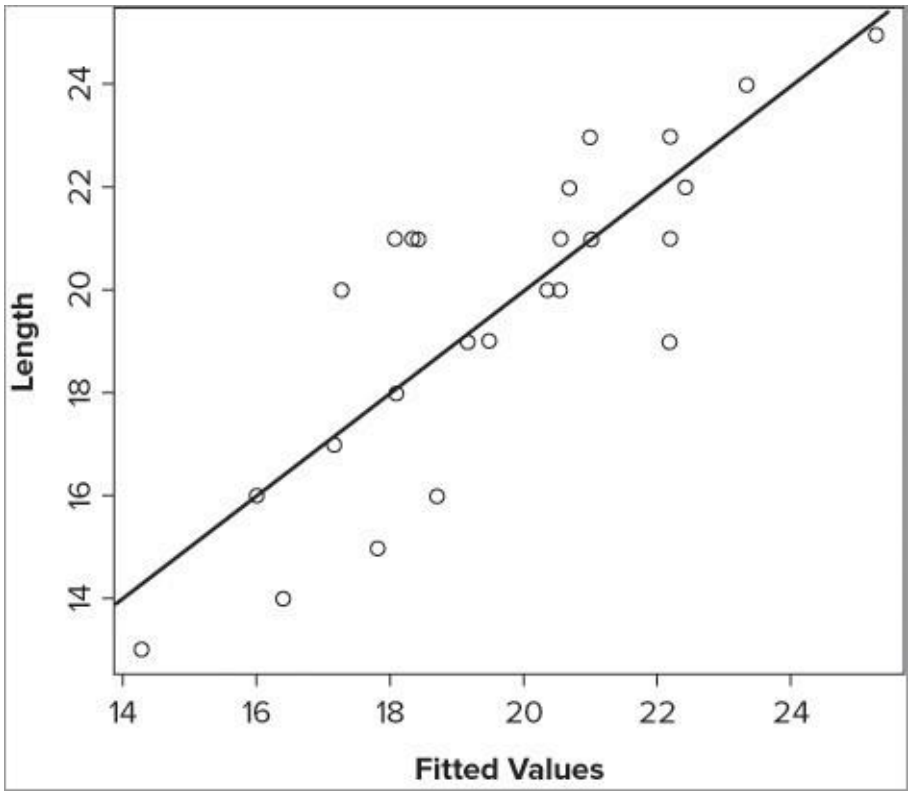
Summary of Fit

If your regression model has a single predictor variable, you can plot the response against the predictor and see the regression in its entirety. If you have two predictor variables, you might attempt a 3-D graphic, but if you have more variables you cannot make any sensible plot. The diagnostic plots that you saw previously are useful, but are aimed more at checking the assumptions of the method than showing how "good" your modelis.

You could decide to present a plot of the response variable against the most significant of the predictor variables. Using the methods you have seen previously you could produce the plot and add a line of best-fit as well as confidence intervals. However, this plot would tell only part of the story because you would be ignoring other variables.

One option to summarize a regression model is to plot the response variable against the fitted values. The fitted values are effectively a combination of the predictors. If you add a line of best-fit, you have a visual impression of how good the model is at predicting changes in the response variable; a better model will have the points nearer to the line than a poor model. The approach you need to take is similar to that you used when fitting lines to curvilinear models. In the following activity you try it out for yourself and produce a summary model of a multiple regression with two predictor variables.

Figure 10-11:



Summary

- You use the lm()command for linear modeling and regression.

- The regression does not have to be in the form of a straight line; logarithmic and polynomial regressions are possible, for example.
- Use the formula syntax to specify the regression model.
- Resultsobjectsthatarisefromthelm()commandincludecoefficients,fitted values,andresiduals.Youcanaccesstheseviaseparatecommands.
- You can build regression models using forward or backward stepwise processes.
- You can add lines of best-fit using the abline() command if they are straight.
- You can add curved best-fit lines to plots using the spline() and lines() commands.
- Confidence intervals can be calculated and plotted onto regression plots. You can
- produce diagnostic plots easily using the plot()command.

Exercises

You can find the answers to these exercises in Appendix A.

1. Look at the mtcars data that are built into R. You saw these data earlier when you used the mpg variable as the response and built a regression model. Compare three linear models: use the wt and cyl variables by themselves andtogether.

2. Earlieryoulookedatthemtcarsdataandbuiltaregressionmodel;thewt variable was the best single predictor variable in the regression. How can you plot the relationship between mpg and wt and include a line of best-fit and99-percentconfidenceintervals?

3. The regression model you created earlier was a forward, additive model; that is, you added terms one after another. Take the mtcars data again and create a backward deletion model. How does this compare to the earlier (forward)model?

4. How can you compare the forward and backward regression models for the mtcarsdata?

5. Now you have created a range of regression models for the mtcars data. How can you produce an overall model plot that shows the line of best-fit and confidencebands?

Chapter 11

More About Graphs

What You Will Learn In This Chapter:

- How to add error bars to existing graphs How to
- add legends to plots
- How to add text to graphs, including superscripts and subscripts
- How to add mathematical symbols to text on graphs How to add
- additional points to existing graphs
- How to add lines to graphs, including mathematical expressions How to
- plot multiple series on a graph
- How to draw multiple graphs in one window
- How to export graphs to graphics files and other programs

Previously, you saw how to make a variety of graphs. These graphs were used to illustrate various results (for example, scatter plots and box-whisker plots) and also to visualize data (for example, histograms and density plots). It is important to be able to create effective graphs because they help you to summarize results for others to see, as well as being useful diagnostic tools. The methods you learned previously will enable you to create many types of useful graphs, but there are many additional commands that can elevate your graphs above the merelyadequate.

In this chapter you look at fine-tuning your plots, adding new elements, and generally increasing your graphical capabilities. You start by looking at adding various elements to existing plots, like error bars, legends, and marginal text. Later you see how to create a new type of graph, enabling you to plot multiple series on onechart.

Error Bars

Error bars are an important element in many statistical plots. If you create a box- whisker plot using the boxplot() command, you do not need to add any additional information regarding the variability of the samples because the plot itself contains the information. However, bar charts created using the barplot() command will not show sample variability because each bar is a single value— for example, mean. You can add error bars to show standard deviation, standard error, or indeed any information you like by using the segments()command.

Using the segments() Command for Error Bars

The segments() command enables you to add short sections of a straight line to anexistinggraph.Thebasicformofthecommandisasfollows:

segments(x_start, y_start, x_end, y_end)

The command uses four coordinates to make a section of line; you have x, y coordinates for the starting point and x, y coordinates for the ending point. To see the use of the segments() command in a bar chart perform the following steps.

1. You first need some data. Use the following a data frame with three samplecolumns:

```
> bf
      Grass Heath
Arable 1      3       6
       19
2      4      7       3
3      3      8       8
4      5      8       8
5      6      9       9
6      12     11      11
7      21     12      12
8      4      11      11
9      5      NA      9
10     4      NA      NA
11     7      NA      NA
12     8      NA      NA
```

2. Plot the means of these three samples, and for error bars use standard error. Start by creating an object containing the mean values; you can use the apply() command in thiscase:

```
> bf.m = apply(bf, 2, mean, na.rm = TRUE)
> bf.m
    Grass      Heath     Arable
6.833333  9.00000010.000000
```

3. Notice that you had to use the na.rm = TRUE instruction to ensure that you removed the NAitems.

4. You now need to get the standard error so you can determine the length of each error bar. There is no command for standard error directly, so you need to calculate using *standard deviation sqrt(n)*. Start by getting the standard deviations likeso:

```
> bf.sd = apply(bf, 2, sd, na.rm = TRUE)
> bf.sd
    Grass      Heath    Arable
5.131601 2.1380904.272002
```

5. WhenyouhaveNAitems,youneedaslightlydifferentapproachtogetthe numberofobservationsbecausethelength()commanddoesnotusena.rm asaninstruction.Theeasiestwayistocalculatethecolumnsumsanddivide bythecolumnmeansbecause*xbar=sum(x)/n:*

```
> bf.s = apply(bf, 2, sum, na.rm = TRUE)
> bf.s
 Grass   HeathArable82
             72       90
> bf.l = bf.s / bf.m
> bf.l
 Grass   HeathArable12
             8        9
```

6. Now that you have the number of replicates, carry on and determine the standarderrors:

```
> bf.se = bf.sd / sqrt(bf.l)
> bf.se
    Grass     Heath    Arable
1.481366 0.7559291.424001
```

7. You now have all the elements you require to create your plot; you have the mean values to make the main bars and the size of the standard errors to create the error bars. However, when you draw your plot you create it from the mean values; the error bars are added afterward. This means that the y- axis may not be tall enough to accommodate both the height of the bars and the additional error bars. You should check the maximum value you need, and then use the ylim instruction to ensure the axis is long enough:

```
> bf.m + bf.se
     Grass     Heath     Arable
 8.314699  9.75592911.424001
```

8. You can see that the largest value is 11.42; this knowledge now enables you to scale the y-axis accordingly. If you want to set the scale "automatically,"youcandeterminethesinglevaluelikeso:

```
> max(bf.m + bf.se)
[1] 11.424
```

9. Tomakethisevenbetter,roundthisvalueuptothenearesthigherinteger:

```
> bf.max = round(max(bf.m + bf.se) + 0.5, 0)
>
bf.max
[1] 12
```

10. You can now make your bar chart and scale the y-axisaccordingly:

```
> bp = barplot(bf.m, ylim = c(0, bf.max))
```

The graph that results looks like Figure 11-1.

You can add axis labels later using the title() command. Notice that you gave your plot a name—you will use this to determine the coordinates for the error bars. If you look at the object you created, you see that it is a matrix with three values that correspond to the positions of the bars on the x-axis likeso:

```
> bp
      [,1]
[1,]  0.7
[2,]  1.9
[3,]  3.1
```

In other words, you can use these as your x-values in the segments() command.Youaregoingtodrawlinesfromtoptobottom,startingfromavalue that is the mean plus the standard error, and finishing at a value that is themean minusthestandarderror.Youaddtheerrorbarslikethis:

```
> segments(bp, bf.m + bf.se, bp, bf.m - bf.se)
```

The plot now has simple lines representing the error bars (see Figure 11-2).

The segments() command can also accept additional instructions that are relevant to lines. For example, you can alter the color, width, and style using instructions you have seen

before (that is, col, lwd, and lty).

Youcanaddcross-barstoyourerrorbarsusingthesegments()commandlike so:

```
> segments(bp - 0.1, bf.m + bf.se, bp + 0.1, bf.m + bf.se)
> segments(bp - 0.1, bf.m - bf.se, bp + 0.1, bf.m – bf.se)
```

The first line of command added the top cross-bars by starting a bit to the left of each error bar and drawing across to end up a bit to the right of each bar. Here you used a value of ±0.1; this seems to work well with most bar charts. The second line of command added the bottom cross-bars; you can see that the command is very similar and you have altered the + sign to a – sign to subtract the standard error from the mean.

You can now tidy up the plot by adding axis labels and "grounding" the bars; the full set of commands is as follows:

```
> bp = barplot(bf.m, ylim = c(0, bf.max))
> segments(bp, bf.m + bf.se, bp, bf.m - bf.se)
> segments(bp - 0.1, bf.m + bf.se, bp + 0.1, bf.m + bf.se)
> segments(bp - 0.1, bf.m - bf.se, bp + 0.1, bf.m - bf.se)
> box()
> title(xlab = 'Site - Habitat', ylab = 'Butterfly abundance')
> abline(h = seq(2, 10, 2), lty = 2, col = 'gray85')
```

The box() command simply adds a bounding box to the plot; you could have made a simple "grounding" line using the following:

```
abline(h = 0)
```

You add simple axis titles using the title() command, and finally, use abline()to make some gridlines. The finished plot looks like Figure 11-3.

Figure 11-1:

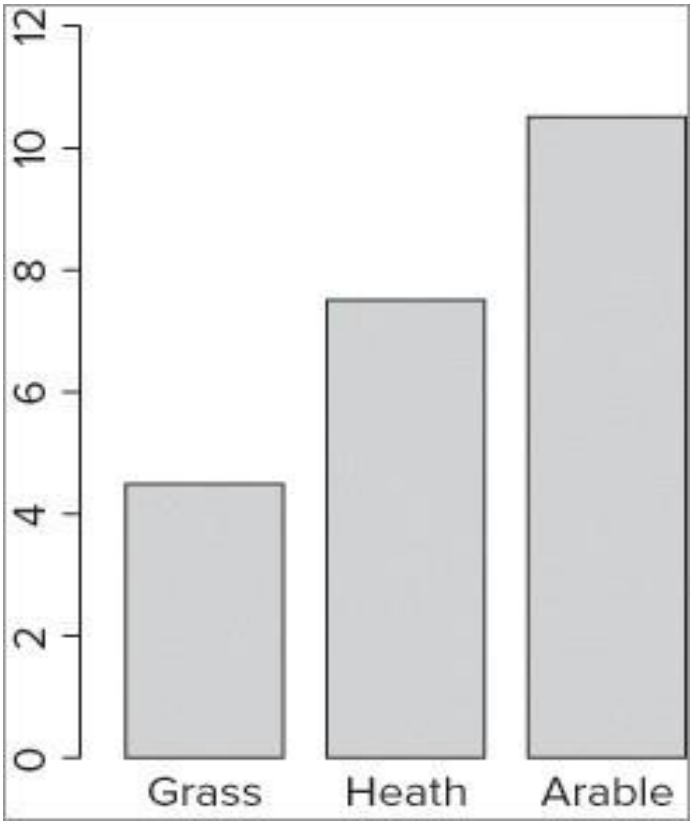


Figure 11-2:

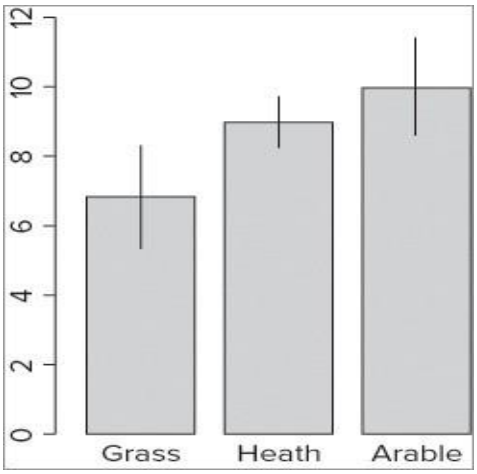
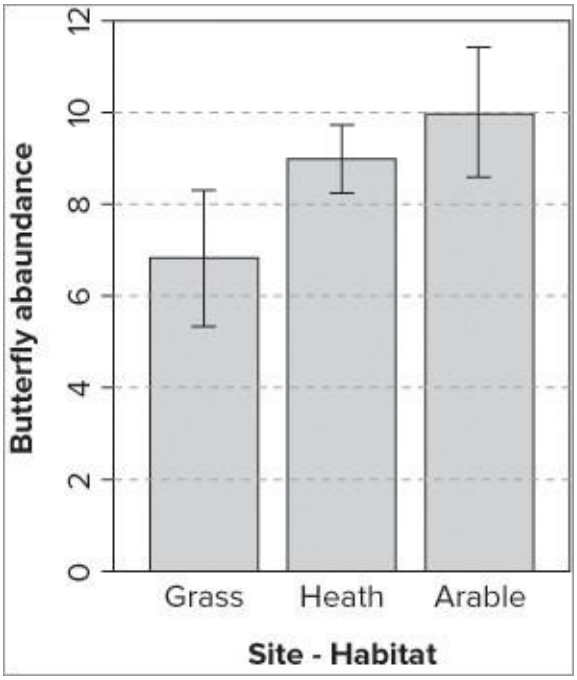


Figure 11-3:



TryIt Out: Create and Add Error Bars to a Bar Chart

Use the grass data object from the Beginning.RData file for this activity. It has two columns of data: rich is the response variable and graze is the predictor.

1. Make a result object that contains mean values for the two levels of the responsevariable:

```
> grass.m = tapply(grass$rich, grass$graze, FUN = mean)
```

2. Now determine the standarderror:

```
> grass.sd = tapply(grass$rich, grass$graze, FUN = sd)
> grass.l = tapply(grass$rich, grass$graze, FUN = length)
> grass.se = grass.sd / sqrt(grass.l)
```

3. Workoutthemaximumheightthatthebarpluserrorbarwillreach:

```
> grass.max = round(max(grass.m + grass.se)+0.5, 0)
```

4. Createthebasicbarchartandaddaxislabels:
```
> bp = barplot(grass.m, ylim = c(0, grass.max))
> title(xlab = 'Treatment', ylab = 'Species Richness')
```
5. Drawinthebasicerrorbars:
```
> segments(bp, grass.m + grass.se, bp, grass.m - grass.se, lwd =2)
```
6. Addhatstotheerrorbars:
```
> segments(bp - 0.1, grass.m + grass.se, bp + 0.1, grass.m + grass.se, lwd =2)
> segments(bp - 0.1, grass.m - grass.se, bp + 0.1, grass.m - grass.se, lwd =2)
```

How It Works

Because you have a response variable and a predictor variable, you must use the tapply() command to calculate the means and the values required for the error bars. Because you have no NA items, you do not need to modify the command. You can extract the number of replicates using the length() command as the FUN instruction. You need the mean values to make the basic plot and the standard error, calculated from the standard deviation and the square root of the number in each sample.

The error bars may be "too tall" for the axis, so you need to add the standard error to the mean to work out the limit of the y-axis. You can now use the barplot() command along with the ylim instruction to make the plot and ensure that the y-axis is the right length. The plot is given a name so that you can determine the x-coordinate of the bars. The segments() command draws the error bars from a point that lies one standard error above the mean to a point one standard error below.

Using the arrows() Command to Add Error Bars

You can also use the arrows() command to add your error bars; you look at this command a little later in the section on adding various sorts of line to graphs. The main difference from the segments() command is that you can add the "hats"totheerrorbarsinonesinglecommandratherthanseparatelyafterward.

Adding Legends to Graphs

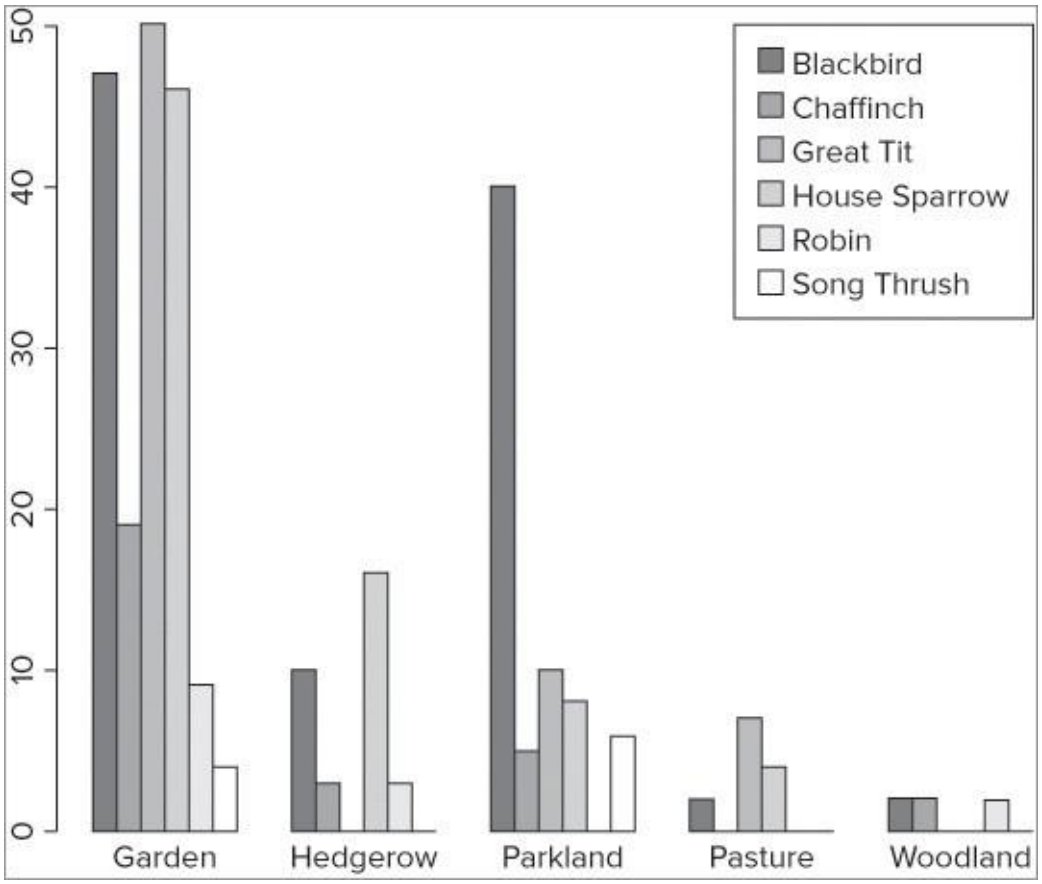
If you have more than one series of data to plot as a bar chart, you may need to create a legend to add to your plot. There is a legend() command to create legends and add them to existing plots. You can also use legend as an instruction in some plot types, including the barplot() command. In the following example you see a data matrix; the barplot() command is used to create a bar chart and include a legend:

```
> 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

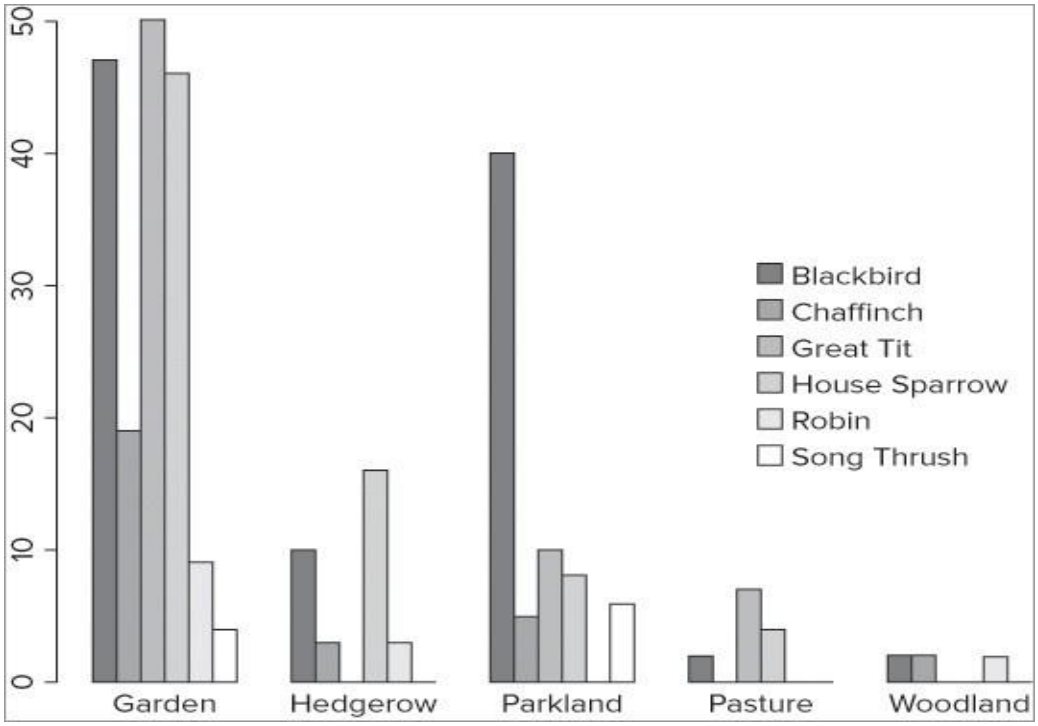
```
> barplot(bird, beside = TRUE, legend = TRUE)
```

Figure 11-4:



The new graph looks like Figure 11-5.

Figure 11-5:



You include all the instructions for the legend() part inside a list() command. These instructions can alter the colors displayed, the plotting characters, the text, and other items. You see some of these instructionsshortly.

Color Palettes

The barplot() example (refer to Figure 11-5) used default colors (gamma- corrected gray colors, in this case), but you can alter the colors of the bars by specifying them explicitly or via the palette() command. If you use the palette()command without any instructions, you see the current settings:

```
> palette()
[1]"black"   "red"     "green3"   "blue"    "cyan"    "magenta"
"yellow"
[8] "gray"
```

These colors will be used for plots that do not have any special settings (many do). It is useful to be able to set the colors so that you can match up the colors of the bars of a barplot()to the colors in a legend().

In the current barplot() example you have six series of data (corresponding to the six rows), so you require your customized palette to have six colors. You have various options; in the following example you create six gray colors:

```
> palette(gray(seq(0.5, 1, len = 6)))
> palette()
[1]"#808080""gray60"   "gray70"   "gray80"   "#E6E6E6""white"
```

In this case you start with a 50 percent gray (0.5) and end up with white (1); you split the sequence into six parts (len = 6). You can reset to the default palette at any time using

thefollowing:

```
> palette('default')
```

Now, whenever you need your sequence of colors you can use palette() to utilize them like so:

```
> palette(gray(seq(0.5, 1, len = 6)))
> barplot(bird, beside = TRUE, col = palette(), legend = TRUE, args.legend = list(bty =
'n'))
```

You do not need to specify the colors of the legend in this case because you have specified the palette() colors for the plot. However, if you use the legend() command separately, you do need to specify the fill colors for the legend boxes like so:

```
> barplot(bird, beside = T, col = palette())
> legend(x = 'topright', legend = rownames(bird), fill = palette())
```

You see more about legends in the next section. Table 11-1lists some other color palette commands you canuse.

Table 11-1: Color Palettes

Color palette command	Explanation
rainbow(n, alpha = 1, start = 0, end	Sets rainbow colors (red, orange, yellow, and so on); n = the number of colors required;alpha=thetransparency(1issolid);start=beginningcolor(1=red,1/
heat.colors(n,	Sets reds and oranges as the colors.
terrain.colors(n,	Uses a range of colors starting with greens, then yellow and tan.
topo.colors(n,	Uses “topographic” colors, starting with blues, then greens and yellows.
cm.colors(n,	Uses blues and pinks.
gray(levels)	Setsgrayswhere0=blackand1=white.Usuallyavectoroflevelsisgiven.For example,seq(0.5,1,len=6)producingsixgraycolorsfrom50percentto white.

You can also specify the palette()colors explicitly like so:

```
> palette(c('blue', 'green', 'yellow', 'pink', 'tan', 'cornsilk'))
> palette()
[1]"blue"          "green"          "yellow"          "pink"          "tan"
"cornsilk"
```

Placing a Legend on an Existing Plot

As you saw previously, you can create a legend as part of a barplot() command. However, with

most graphs the legend must be added separately after the main plot has been drawn. The legend() command is what you need to do this.

When placing a legend on an existing plot you need to control the colors used so that you can match up the plot colors with the legend colors. In the following example, you create a gray palette using six shades of gray to match the number of bars you will get in each category using the followingcommands:

```
> palette(gray(seq(0.5, 1, len = 6)))
> barplot(bird, beside = TRUE, col = palette())
```

Youseeinthesecondcommandtheinstructionstoplotthebarchartusingthe palette() colors. Now you can use the legend() command to add a legend. Thegeneralformofthelegend()commandisasfollows:

```
legend(x, y = NULL, legend, fill = NULL, col = par('col')
```

Thefirstpartdefinesthexandycoordinatesofthetop-leftpartofthelegend, You can specify a single value as text, for example, "topright". The legend= partisavectorcontainingthenamestoappearinthelegend.Thefill=partis usedwhenyouhaveabarorpiechartandwanttoshowthebarorslicecolorby the names; it creates a simple block of color. The col = part is used when you haveplotswithpointsand/orlines.

In the current example, you place the legend at the top-right corner of the graph like so:

```
> legend(x = 'topright', legend = rownames(bird), fill = palette(), bty = 'n')
```

You set the names of the rows to appear as your legend text and use the palette() colors to become the colored boxes beside each name. Finally, you use the bty = 'n' instruction to set the border type to "none" (the default is "o", presumably for"on").

Inthiscasethegraphlooksexactlythesameastheoneyoudrewearlier(refer to Figure 11-4) except that the legend has no border. You can use all the legend() instructions as part of a barplot() by setting legend = TRUE andthen using args.legend = list(...). You replace the contents of the brackets with the instructions as required; the instructions are summarized in Table 11-2.

Table 11-2: Legend Instructions/Parameters

Instructio	**Explanation**
legend(...)	
x =	Thexcoordinateforthetop-leftofthelegendbox.Ifyisnotspecified,youcanuseoneof the following text strings: 'topright', 'topleft', 'left', 'right', 'top', 'bottom', 'bottomright','bottomleft','center'.Thesemaybeabbreviated.
y =	The ycoordinate for the top left of the legend box.
legend	A vector of names to be used as the text for the legend.
fill =	Ifspecified,thiscausesblockstoappearbesidethelegendnamesfilledinthecolor(s) specified.
col	Used to set colors for lines or points if specified.
bty	The box/border type for the legend; bty = 'n' sets the box/border to "none" and bty = 'o'

pch	Theplottingcharactertouseinthelegendbesidethenames;thesewillbesettothecolor specifiedinthecol=instruction.
lty	Sets the line type if appropriate.
lwd	Sets the line width if appropriate.

A host of additional instructions can be used with the legend() command; look at the help entry via the help(legend) command. You see a few more examples later when you look at the matplot()command.

Adding Text toGraphs

Adding text to a graph can improve its readability enormously, and generally turn it into an altogether more useful object. You have already seen a few examples of how to add text to an existing plot; the most obvious one being the axis labels (using xlab and ylab). Text can also be used within the plot area, to show statistical results, forexample.

Here you look at ways to add text to the main plotting area, as well as means to customize your axis labels in various usefulways.

Making Superscript and Subscript Axis Titles

So far the axis titles that you have used have all been plaintext, but you may

want to use superscript or subscript. The key to achieving this is the expression() command; this takes plaintext and converts it into a more complex form. You can use the expression() command to create mathematical symbols too, which you seelater.

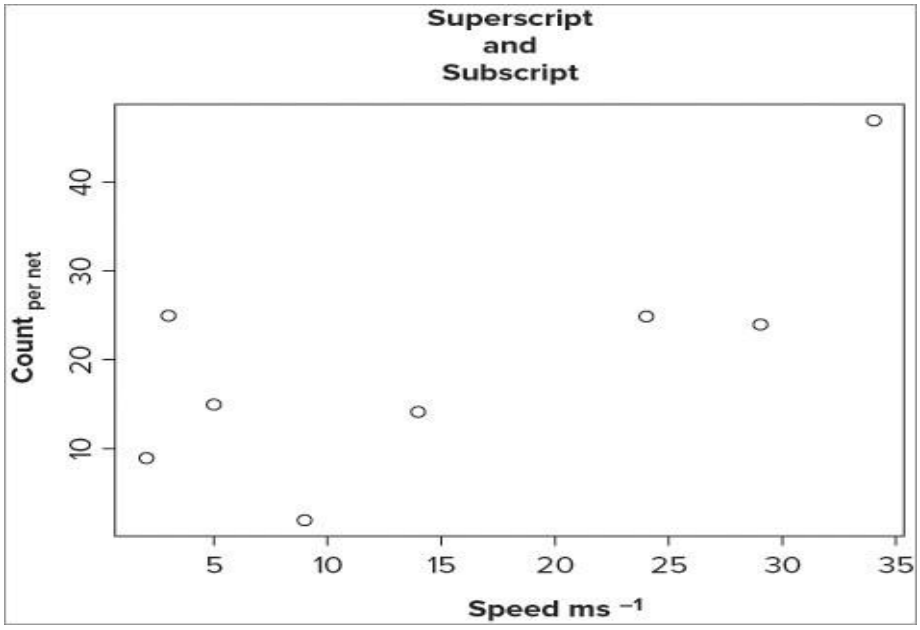
You can use the expression() command directly as part of your plot() command or you can create your text label separately; in any event you use rules similar to "regular expressions" to generate the output you require. In the following example you create a simple scatter plot and then add titles to the axes using the title()command; this makes it easier to see:

```
> plot(count ~ speed, data = fw, xlab = "", ylab = "")
> title(xlab = expression('Speed ms'^-1))
> title(ylab = expression('Count '['per net']))
> title(main = 'Superscript\nand\nSubscript')
```

The plot() command simply draws a scatter plot; you suppress the axis titles by using double quotes. In expression() syntax a caret symbol (^) means "createsuperscript,"sothefirstpartofyourlabelisinregularquotesandthe ^-1part that follows creates a superscript $^{-1}$. To create subscript you enclose the partyouwantinsquarebrackets;notethatyouhavetwosetsofquotedtext,one outsidethe[]andtheotherinsidethe[].

Although not strictly an axis label, the example included a main title to illustrate that you can create newlines using \n; in this example, the \n is ignored as text and the text following it appears on a fresh line. The graph that results from this looks like Figure 11-6.

Figure 11-6:



You do not always need to put your text in quotes like regular axis labels because the expression() command converts what you want into a text string. However, the command does not like spaces, and in this case you required Speed ms^{-1} as a label. You can use the * to link the items together; the following example would have produced the same text as before:

```
> title(xlab = expression(Speed*' '*ms^-1))
```

This time you use a pair of quotes with a single space to split the two elements. You also used quotes for the subscripted label; you cannot avoid them in the ['per net'] part, because the space needs to be present. You could, however, drop the quotes from the first part; the following would give you the same result asbefore($Count_{pernet}$):

```
> title(ylab = expression(Count['per net']))
```

Better still would be to avoid quotes altogether and use the tilde (~) symbol. This takes the place of spaces in the expression() command. In the current example, therefore, you would use the followingcommands:

```
> title(xlab = expression(Speed ~ ms^-1))
> title(ylab = expression(Count[per ~ net]))
```

So, the spaces you see in the commands are actually ignored and it is the ~ symbols that produce the spaces in the final text labels.

If you require your labels to be in another typeface, such as italic, you need to alter the expression() text itself by adding an extra part. You have several options, as shown in Table 11-3.

Table 11-3: Modifying the Font in a Text expression()

Command	Explanation
plain(text)	The text in the brackets is plaintext.
italic(text)	Makes the text in the brackets italic font.
bold(text)	Creates bold font.
bolditalic(text)	Sets the font for the following text to bold and italic.

For the example you have here, your x- and y-axis labels could be set to italics in the followingmanner:

```
> title(ylab = expression(italic(Count['per net'])))
> title(xlab = expression(italic(Speed*' '*ms^-1)))
```

Itispossibletobuildupquitecomplexlabelsinthismanner,butitisalsoeasy to make a mistake! For this reason it can be helpful to create objects to hold the expression()labels:

```
> xl = expression(italic(Speed ~ ms^-1))
> yl = expression(italic(Count[per ~ net]))
> plot(count ~ speed, data = fw, xlab = xl, ylab = yl)
```

You can also use the expression() command to create mathematical symbols and expressions; you return to these shortly.

Orienting the Axis Labels

For the most part you have used labels on your plots that were oriented in a single direction:

horizontal to the axis. You can alter this using the las = instruction. Recall that you used this when looking at the results of a Tukey HSD test. The instruction is useful in helping to fit in labels that would otherwise overlap. Table 11-4 summarizes the options for the lasinstruction.

Table 11-4: Options for the las() Command for Axis Label Orientation

Comman	Explanation
las = 0	Labels always parallel to the axis (the default).
las = 1	All labels horizontal.
las = 2	Labels perpendicular to the axes.

If you rotate the axis labels, it is possible that they may not fit into the plot margin, and you may need to alter this to make room. This is the subject of the followingsection.

Making Extra Space in the Margin for Labels

You can alter the margins of a plot with a call to the mar = instruction. You cannot set the margins "on the fly" while you are creating a plot so you must use the par() command before you start. The general form of the mar() instruction is asfollows:

mar = c(bottom, left, top, right)

You use numeric values to set the size of the margins (as lines of text). The default values are:

mar = c(5, 4, 4, 2) + 0.1)

Setting Text and Label Sizes

You can set the sizes of text and labels using the cex = instruction. You can use several subsettings, as summarized in Table 11-5.

Table 11-5: Options for Setting Character Expansion

Comman	Explanation
cex = n	Sets the magnification factor for the plotting characters (ortext).Ifn=1,"normal"size is
cex.axis	The magnification factor for axis labels; for example, units on y-axis.
cex.lab	Magnification factor for main axis labels.
cex.nam	Sets the magnification factor for names labels; for example, bar category labels.
cex.mai	The magnification factor for the main plot title.

Adding Text to the Plot Area

Adding text to an existing graph can be useful for many reasons; you may need to label

various points or state a statistical result, for example. You can add text to an existing graph by using the text() command. You specify the x and y coordinates for your text and then give the text that you require; this can be in the form of text in quotes or an expression(). The general form of the command is asfollows:

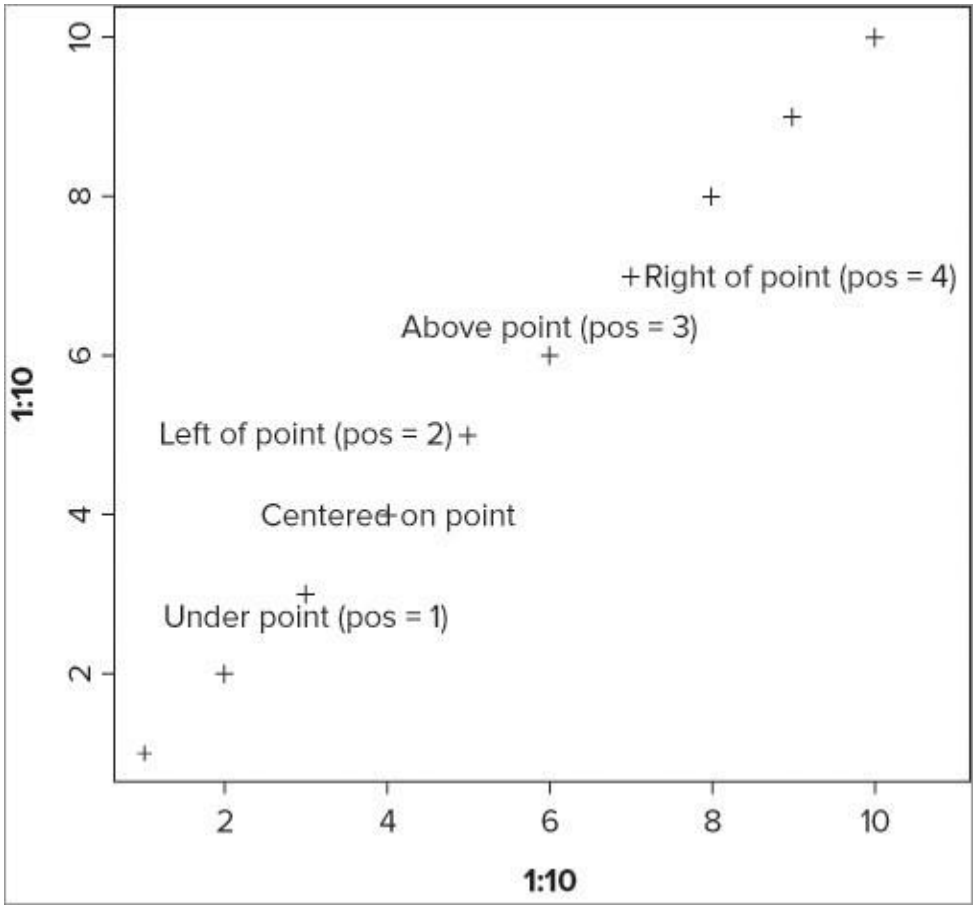
text(x, y, labels = , ...)

The text you create is effectively centered on the x, y coordinates that you supply;youcanalterthisbyusingthepos=instruction.Thefollowingexample showstherangeofoptions:

Table 11-6: Options for Altering Text Appearance when Added to a Graph

Instructio	**Explanation**
col	Sets the text color, usually with a simple name. For example, ‘gray50’.
cex = n	The character expansion factor >1 makes text larger and <1 makes it smaller.
font = n	Sets the font type; 1 = plain (default), 2 = bold, 3 = italic, 4 = bold italic.
pos = n	Thepositionofthetextrelativetothecoordinates;1=below,2=totheleft,3=above,4= totheright.Ifmissing,thetextiscenteredonthecoordinates.
offset	An additional offset in character widths (default = 0.5).
srt = n	The rotation of the text in degrees, thus srt = 180produces upside down text.

Figure 11-7:



Knowing exactly where to place the text can be a bit hit-or-miss and involves some trial and error. However, it is also possible to determine the appropriate coordinates by using the locator() command. You have two possibilities. You can simply use the locator() command to find coordinates by clicking on a plot:

```
locator(1)
```

You now select the graph window, and when you left-click in the plot area you get the x and y coordinates of the position you clicked; you can then use these in your text() command. Alternatively, you can use locator(1) as part of the text()command:

```
> text(locator(1), 'Text appears at the point you click')
```

After you press Enter the program waits for you to click in the plot window; then the text appears. In this example the text would be centered on the point you click, but you can add the pos =instruction to alter this (refer Table 11-6).

Adding Text in the Plot Margins

You have already seen how to label the main axes using xlim and ylim instructions, but sometimes you need to add text into the marginal area of a graph. You can add text to the plot margins using the mtext() command. The general form of this command is as follows:

```
mtext(text, side = 3, line = 0, font = 1, adj = 0.5, ...)
```

You can use regular text (in quotes) or an expression(), or indeed any object that produces text. The side = instruction sets which margin you want (1 = bottom, 2 = left, 3 = top, 4 = right); the

default is the top (side = 3). You can also offset the text by specifying the line you require; 0 (the default) is at the outer part of the plot area nearest the plot. Positive values move the text outward and negative values move it inward. You can also set the font, color, and other attributes.Thefollowingexampleproducesasimpleplottodemonstrate:

```
> plot(1:10, 1:10)
> mtext('mtext(side = 1, line = -1)', side = 1, line = -1)
> mtext('mtext(side = 2, line = -1, font = 3)', side = 2, line =
-1, font = 3)
> mtext('mtext(side = 3, font = 2)', side = 3, font = 2)
> mtext('mtext(side = 3, line = 1, font = 2)', line = 1, side = 3,
font = 2)
> mtext('mtext(side = 3, line = 2, font = 2, cex = 1.2)', cex =
1.2, line = 2, side = 3, font = 2)
> mtext('mtext(side = 3, line = -2, font = 4, cex = 0.8)', cex =
0.8, font = 4, line =-2)
> mtext('mtext(side = 4, line = 0)', side = 4, line = 0)
```

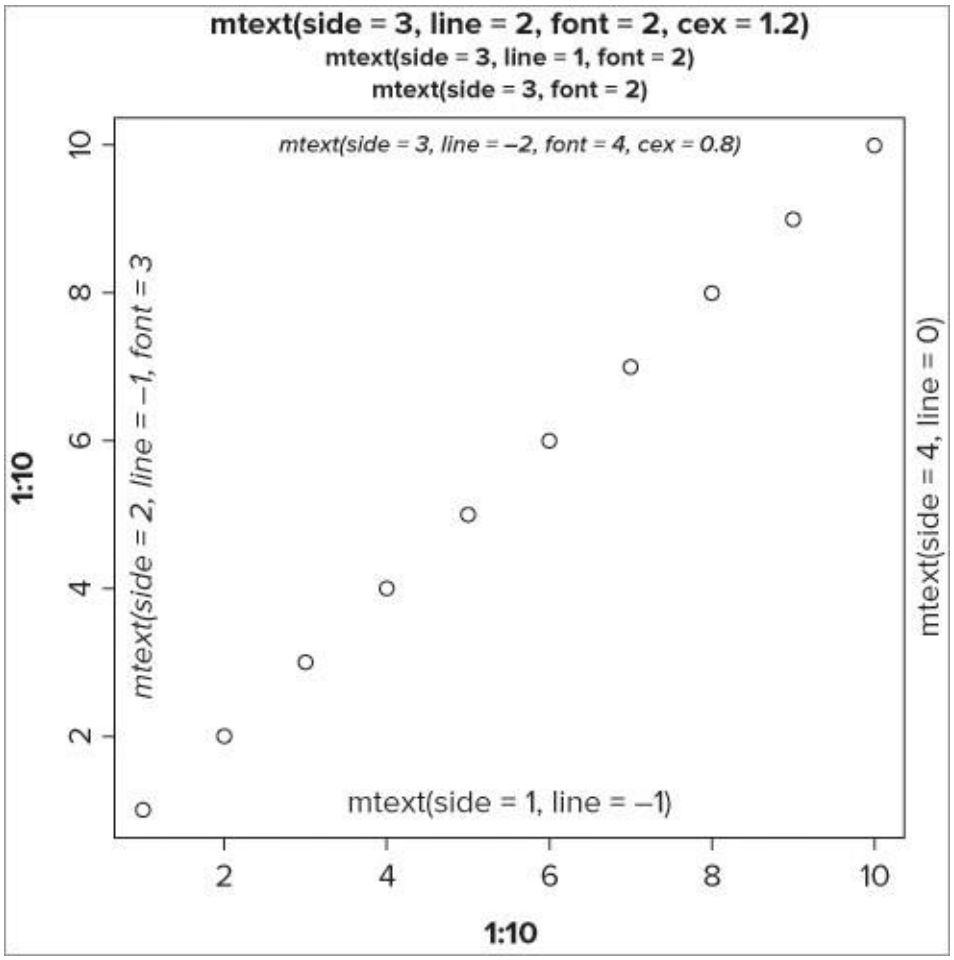
This produces a graph that looks like Figure 11-8.

You can adjust how far along each margin the text appears using the adj = instruction; a value of 0 equates to the bottom or left of the margin, and a value of 1 equates to the top or right of the margin. The default equates to 0.5, the middle of the side. Table 11-7 shows the mainoptions.

Table 11-7: Options for using the mtext() Command

Optio	Explanation
side = 3	Sets the side of the plot window to add the text; 1 = bottom, 2 = left, 3 = top (default), 4 = right.
line	Which line of the margin to use; 0(the default) is nearest to the plot window. Larger values move outward and lower (-ve) values move in to the plot area.
adj	Howfaralongtheaxis/margintoplacetext:0equatestoleftorbottomalignmentand1equates torightortopalignment.
cex	The character expansion factor; values > 1 make text larger and values < 1 make it smaller.
col	The color to use; generally a text value. For example, “gray40”.
font	Sets the font; 1 = plain (default), 2 = bold, 3 = italic, 4 = bold italic.
las	Theorientationofthetextrelativetotheaxis/margin:0=paralleltoaxis(default),1=horizontal, 2=perpendiculartoaxis/margin,3=vertical.

Figure 11-8:



Creating Mathematical Expressions

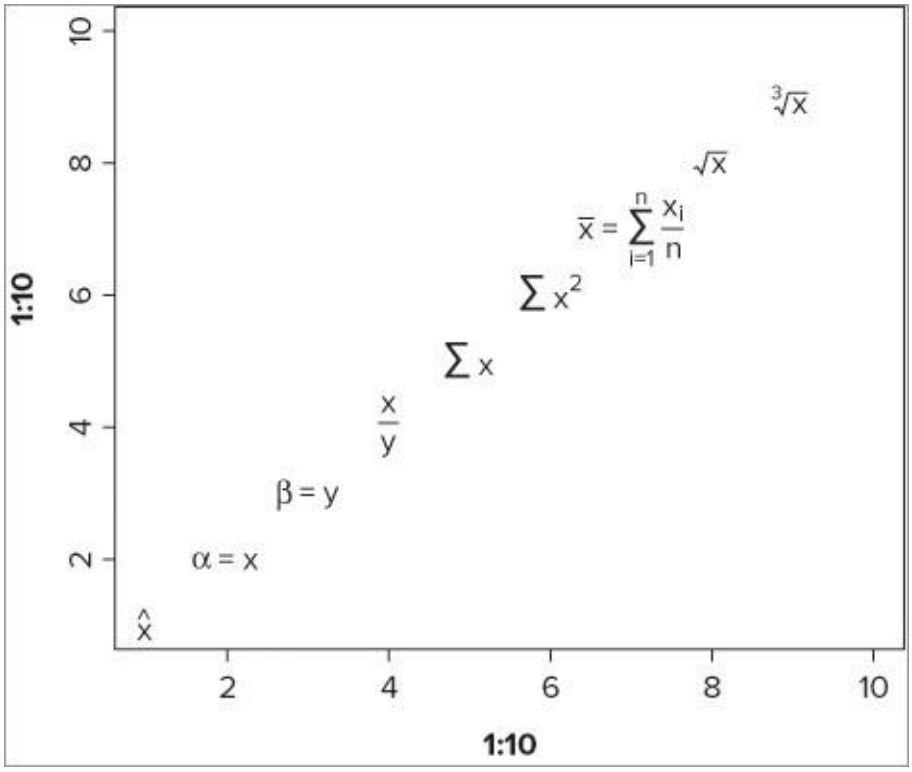
Earlier you saw how to use the expression()command to make superscript and subscript text. The command also enables you to specify complex mathematical expressions in a similar manner. You have lots of options, but the following examples illustrate the most useful ones:

```
> plot(1:10, 1:10, type = 'n')
> opt = par(cex = 1.5)

>   text(1,   1,  expression(hat(x)))
>   text(2,   2,  expression(alpha==x))
>   text(3,   3,  expression(beta==y))
>   text(4,   4,  expression(frac(x, y)))
>   text(5,   5,  expression(sum(x)))
>   text(6,   6,  expression(sum(x^2)))
>   text(7,   7,  expression(bar(x) == sum(frac(x[i],          n),   i==1,   n)))
>   text(8,   8,  expression(sqrt(x)))
>   text(9,   9,  expression(sqrt(x, 3)))
```

par(opt)

This produces something like Figure 11-9.

Figure 11-9:



You begin by creating a blank plot and then reset the cex instruction to produce text 1.5 times larger than normal. The cex instruction could, of course, be used for each text()command, but this way is more efficient.

The first command produces a simple letter x with a hat: ˆx. The second command produces a Greek character. You use two equal signs to get your final text: α = x. The next example uses the Greek letter beta and you end up with β

= y.

The fourth command creates a fraction; you specify the top and the bottom parts of the fraction separated with a comma. The fifth command creates a ∑ character, and you make this more complex in the sixth command by adding a superscript.

The seventh command is quite complicated and uses several elements including a subscript. The final two commands use the square root symbol.

You can alter the character size directly using the cex = instruction and can also control the color via the col = instruction. The typeface must be set from within the expression itself. For example:

expression(italic(x^2))

This command produces x^2, with the whole lot being in an italic typeface. The expression() syntax enables you to produce more or less any text you require; for full details look at help(plotmath). Table 11-8 outlines a few of the options for producing math inside text expressions.

Recall the bar chart you created earlier when you created and added error bars using the segments() command. In the following activity you will recreate the graph and add a variety of text items to it.

Adding Points to an Existing Graph

You may need to add points to an existing graph, for example, if you need to create a scatter plot containing more than one sample. You can add points to an existing plot window by using the points() command. This works more or less like the plot() command, except that you do not create a new graph. The general form of the points() command is as follows:

points(x, y = NULL, type = "p", ...)

You specify the x and y coordinates for your points like you would with any other plot(); this means that you can use a formula, in which case you do not need the y = instruction because the x and y coordinates are taken from the formula. You can also add other graphical instructions that affect the plotting characters; you can alter the size, color, and type for example. You cannot, however, alter the y-axis size or the axis labels because these will only work when you make a new plot. Instead, you use the title() command to make axis titles or the mtext() command. The following example shows the creation of a simple two-series scatter plot. The data are a simple data frame with three columns:

The first column contains the predictor variable (the x-axis) and the next two columns are response variables (two lots of y-axis). You can create a simple scatter plot using the plot() command like so:

> plot(sfly ~ speed, data = fwi, pch = 21, ylab = 'Abundance', xlab = 'Speed')

Notice that you have specified the plotting character and axis titles explicitly. The graph now shows one series of data in the plot window. You can add the second series by using the points() command:

> points(mfly ~ speed, data = fwi, pch = 19)

To differentiate between the two series of data, you use a different plotting character (pch = 19); you can also use a different color or size via the col = and cex = instructions. To add a legend to help the reader see the different series, you can use the legend()command in the following way:

> legend(x = 'topright', legend = c('Stonefly', 'Mayfly'), pch = c(21,19), bty = 'n')

In this case you place the legend at the top right of the plot window and set the legend text explicitly. You also set the plotting characters that appear in the legend to match those used in the plots; if you had used different colors you could set the colors to match in a similar fashion. The commands are repeated in the followingcode:

> plot(sfly ~ speed, data = fwi, pch = 21, ylab = 'Abundance', xlab = 'Speed')
> points(mfly ~ speed, data = fwi, pch = 19)
> legend(x = 'topright', legend = c('Stonefly', 'Mayfly'), pch = c(21,19), bty = 'n')

The final graph looks like Figure 11-10.

In the current example it does not make a lot of sense to join the dots, but it is possible to create points that are joined for use in more appropriate circumstances.

You can use the points() command very much like the plot() command and create a series of joined-up points. When you create a scatter plot that uses only points, you do not need to be concerned about the order in which the items are plotted. In the current example, however, the x-data are unsorted, and if you tried to plot them with connecting lines, you would end up with a mess! You need to sort them in ascending order of the x-variable. Once your data are in the right order, you can proceed with theline-plot.

In the following activity,you reorder the data that you have seen here to produce a line-plot with two series of data.

Figure 11-10:

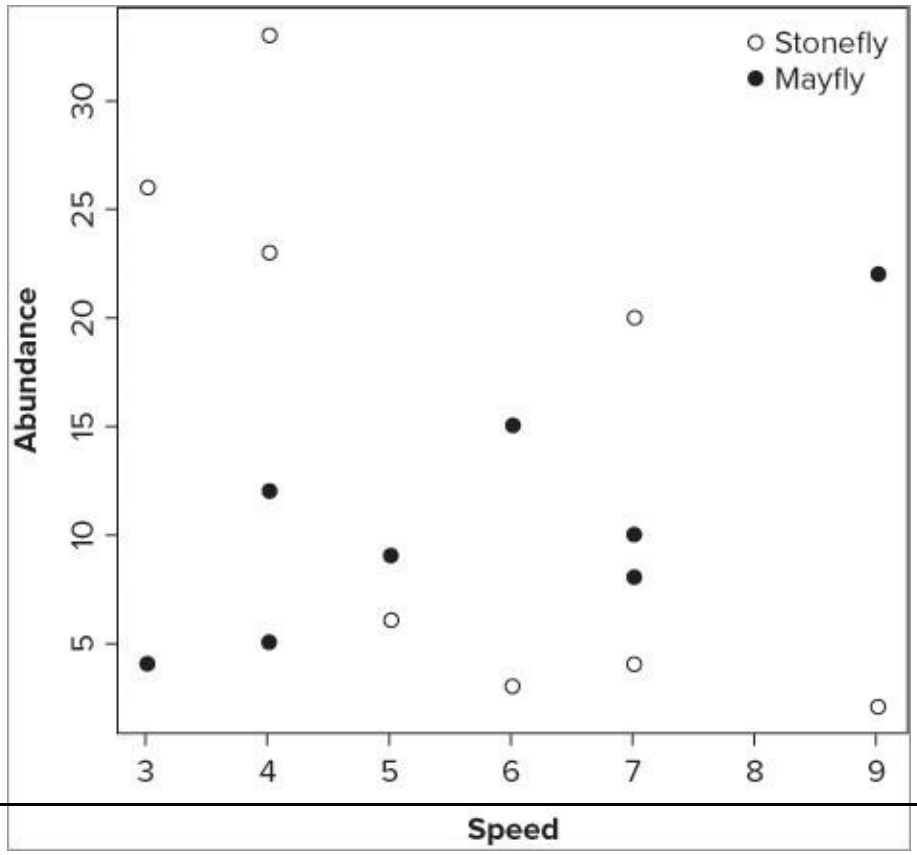
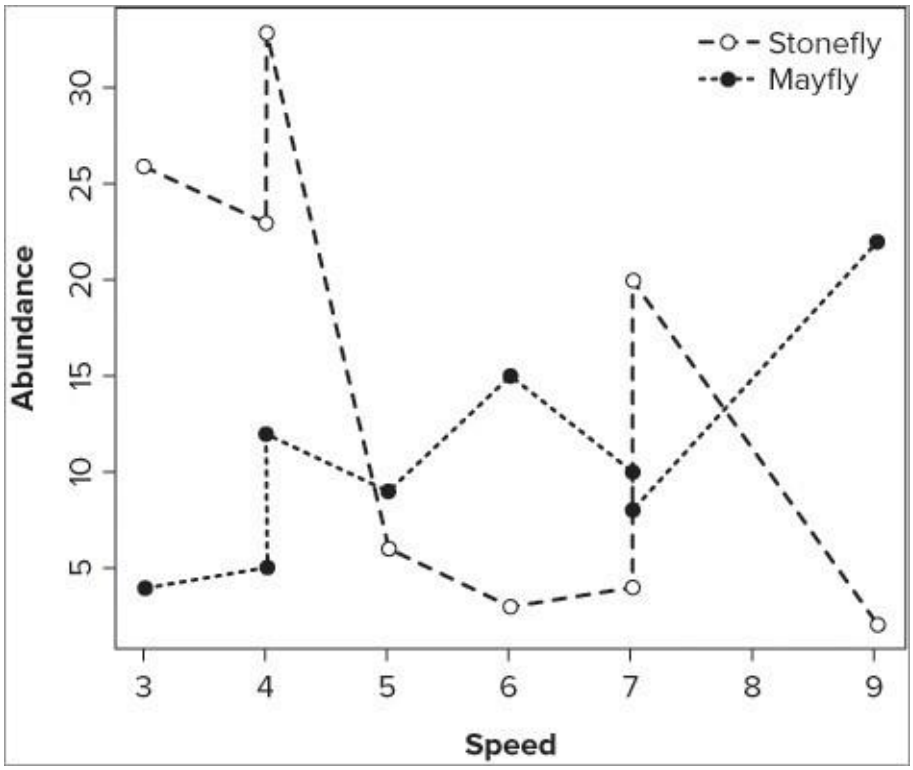


Figure 11-11:



Adding Various Sorts of Lines toGraphs

You have seen some examples of adding lines to existing plots already; you saw how to add straight lines using the abline() command. You also saw how to create lines fitted to results of linear regressions using lines() and the spline() command to make them curved. You also saw how to use the segments() command to add sections of straight line; you used these to create error bars. In this section you review some of these methods and add a few extra ways to draw straight and curved lines onto existinggraphs.

Adding Straight Lines as Gridlines or Best-Fit Lines

You can use the abline() command to add straight lines to an existing plot. The command will accept instructions in several ways; you can use a slope and intercept as explicitvalues:

```
abline(a = slope, b = intercept, ...)
```

The command thus works using the equation of a straight line, that is, $y = a + bx$. You can also specify the equation directly from the result of a linear model:

```
abline(lm(response ~ predictor, data = data), ...)
```

You may already have a linear model and can use the result directly if you have saved it as a named object. Finally, you can specify that you want a horizontal or verticalline:

```
abline(h = value, ...)
abline(v = value, ...)
```

You can alter the display of your line using other instructions; for example, you can change the color, width, style, or thickness. The following activity gives you a chance to explore some of

the possibilities.

Try It Out: Add Various Lines to aGraph

1. Makeascatterplotandbesuretoincludetheorigin:
   > plot(sfly ~ speed, data = fwi, ylim = c(0,30), xlim =c(0,10))
2. Use the result of a linear regression to add a line of best-fit to the graph:
   > abline(lm(sfly ~ speed, data = fwi))
3. Now make a series of horizontalgridlines:
   > abline(h = seq(5, 30, 5), lty = 2, col ='gray50')
4. Make a series of verticalgridlines:
   > abline(v = 1:9, lty = 2, col ='gray50')
5. Addalineusingafixedequation:
   > abline(a = 0, b = 1, lty = 3, lwd =1.8)

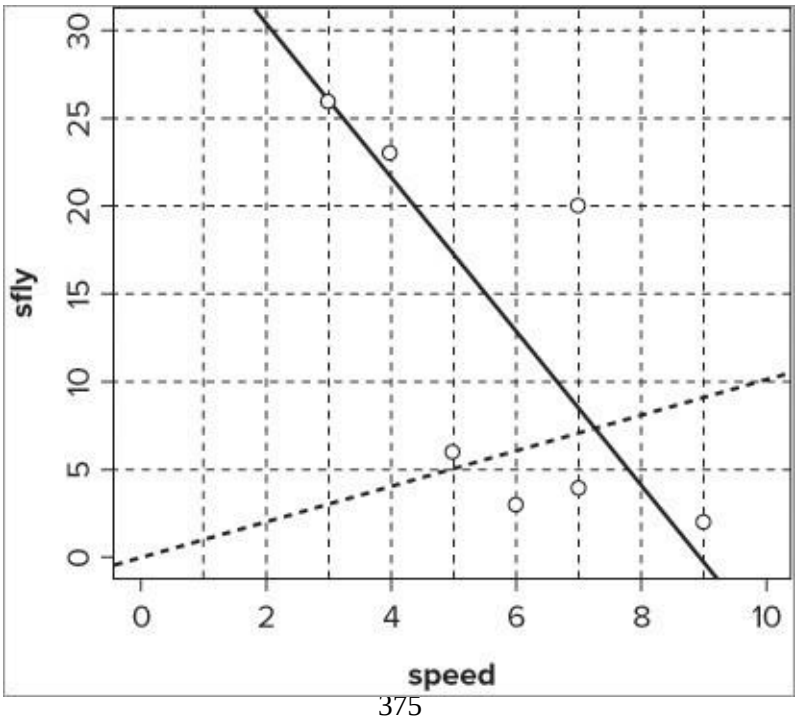
How It Works

The plot() command draws a basic scatter plot (in this case) using simple data you saw earlier. You specify the x- and y-axis limits explicitly because you want to see the origin of the plot (that is, the 0, 0 coordinate). The first abline() command produces a line of best-fit from the linear model between the two variables; this line is unmodified.

The next command creates horizontal lines as gridlines; you use the h = instruction and a sequence of values using seq(). You also make these horizontal lines dashed (lty = 2) and colored. The next line creates vertical lines; this time you use a simple sequence from one to nine. The final command creates a line using explicit intercept and slope values.

The final graph looks like Figure 11-12.

Figure 11-12:



The abline() command is pretty flexible, but it can draw only straight lines; you see how to create curved lines shortly. The command options/instructions for the abline()command are summarized in Table 11-9.

Table 11-9: The abline() Command and its Options

Command/Instructi	Explanation
abline(a = intercept,	Draws a straight line with intercept = a and slope = b.
abline(coef =	Usesalinearmodelresulttocreateinterceptandslope.Alternatively,anyvector containingtwovaluesmaybeinterpretedastheinterceptandslope.
abline(h = value)	Draws horizontal lines at the ycoordinates specified in the value.
abline(v = value)	Draws vertical lines at the x coordinates specified in the value.
lty = n	Setsthelinetype.Linetypescaneitherbespecifiedasaninteger(0=blank,1=solid(default),2=dashed,3=dotted,4=dotdash,5=longdash,6=twodash)or as one of the character strings "blank", "solid", "dashed", "dotted", "dotdash", "longdash", or "twodash", where "blank" uses “invisible lines” (thatis,doesnotdrawthem).
col = color	Sets the line color; usually as a text value. For example, "gray50".
lwd = n	Setsthelinewidthasaproportion;>1makeslinewiderand<1makestheline narrower.

Making Curved Lines to Add to Graphs

You can add curved lines to plots in a variety of ways. The lines() command is especially flexible and allows you to add to existing plots. In a general way, the command takes x and y coordinates and joins them together on your plot. The general form of the command is:

```
lines(x, y = NULL, ...)
```

You can specify the coordinates in many ways. Earlier you used the results of linear modeling to create curved lines of best-fit to both logarithmic and polynomial regressions; you used the original x-values and took the y-values from the fitted()command like so:

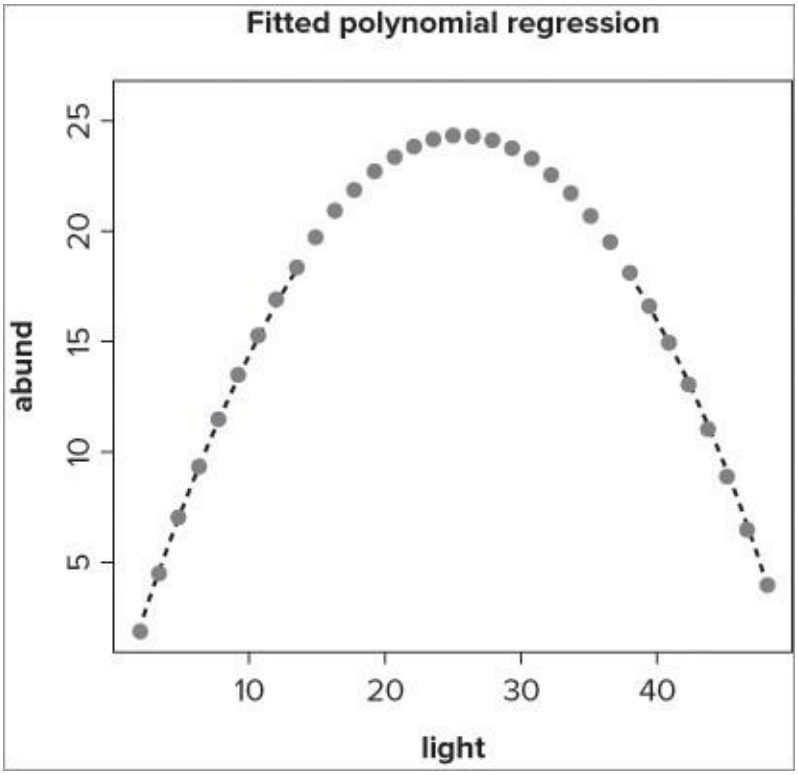
```
> bbel.lm

Call:
lm(formula = abund ~ light + I(light^2), data = bbel)

Coefficients:
(Intercept)                 light         I(light^2)
     -2.00485              2.06010          -0.04029

> plot(abund ~ light, data = bbel)
> lines(bbel$light, fitted(bbel.lm))
```

In this case you specify both x and y coordinates individually, but you could have used a formula instead:

```
> lines(fitted(bbel.lm) ~ light, data = bbel)
```

col = 'gray50')
> title(main = 'Fitted polynomial regression')

Figure 11-13:



Some of the additional graphical instructions you can apply are summarized in Table 11-10.

Table 11-10: Graphical Instructions that Can be Applied to Lines on Graphs

Instructio	Explanation
type	Setsthekindoflinedrawn.type=‘l’(thedefault)producesalineonly,‘b’producesa linesplitbypoints,‘o’producesalineoverlainwithpoints,‘n’suppressestheline.
lty = n	Sets the line type. Line types can either be specified as an integer (0 = blank, 1 = solid (default),2=dashed,3=dotted,4=dotdash,5=longdash,6=twodash)orasoneofthe character strings "blank", "solid", "dashed", "dotted", "dotdash", "longdash", or "twodash",where"blank"uses“invisiblelines”(thatis,doesnotdrawthem).
lwd = n	Sets line width where n = 1 is “normal” width. Values > 1 make line wider and values < 1
col =	Sets line color, usually as a text names. For example, ‘gray50’.
pch = n	Setsplottingcharacter(usediftype=‘b’or‘o’);usuallyspecifiedasanumericalvalue butaquotedcharactermaybeused.Forexample,“+”.

Plotting Mathematical Expressions

You can draw mathematical functions, either as a new plot or to add to an existing one. The plot() command can be pressed into service here and you can also use the curve() command. The general form of both these commands is similar. Following is the curve()command:

curve(expr, from = NULL, to = NULL, n = 101, add = FALSE, type = 'l', log = NULL, ...)

You start by creating an expression to plot. This can be a simple math function (for example, log, sin) or a more complex function that you define. You also need to determine the limits of the plot, that is, the starting and ending values for your x-axis. The command works out the mathematical expression for *n* times, spread across the range of values you specify; *n* is set at 101 by default but you can add extra points to produce a smoother curve. You can add a mathematical curve to an existing plot using the add = TRUE instruction. By default, a line is drawn but you can specify other types, for example, type = 'b', to produce points joined by sections of line. You can also specify a log scale for one or both axes. Finally, you can alter the line type, color, and other parameters much like the other plotting commands you haveseen.

Here are some simple examples:

```
> plot(log)
> plot(log, from = 1, to = 1e3)
> curve(log10, from = 1, to = 1e3, add = TRUE, lty = 3)
```

In the first line you plot a simple logarithm (natural log); the limits of the x- axis are set from 0 to 1 by default. In the second line, you specify that youwant to use 1 and 1000 as the x-axis limits. In the third line you add a curve to the existing plot; you specify log to the base 10 and also set the line type to dotted (that is, lty =3).

If you want a logarithmic axis you can specify this via the log = instruction like so:

```
> curve(log, from = 1, to = 1e3, log = 'x')
```

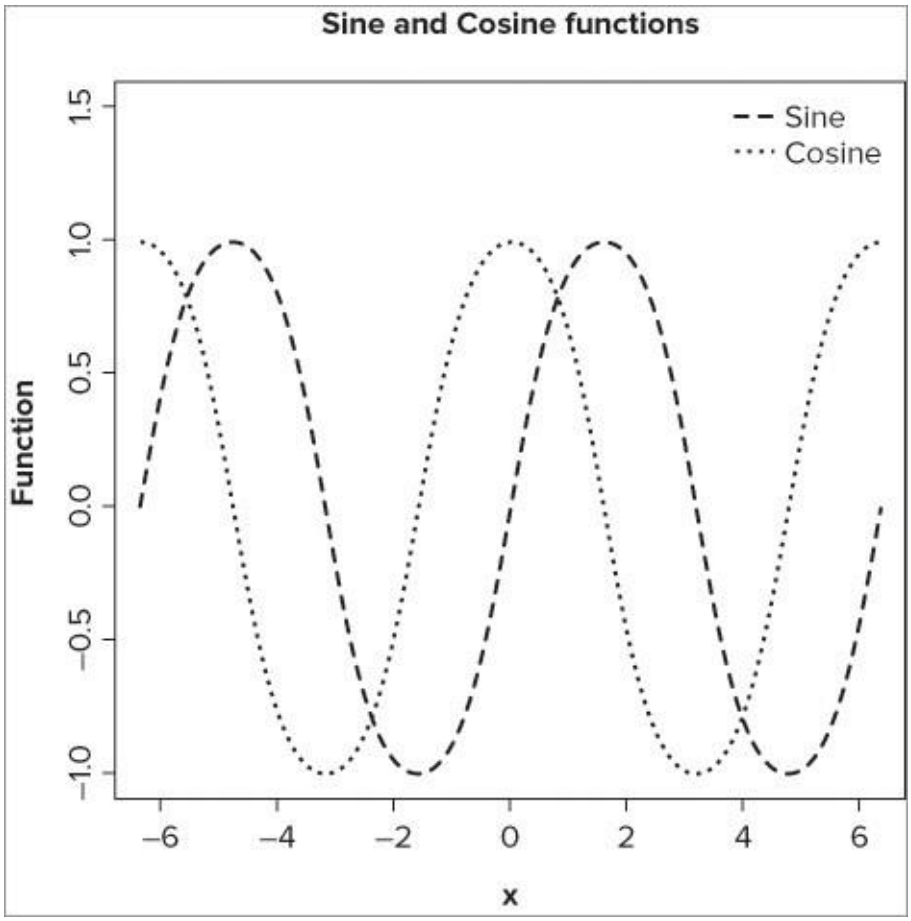
You use a simple text label to say which axis you require to be on the log scale: "" for neither (the default), "x" for just the x-axis, "xy" for both, and "y" for the y-axisonly.

The axis titles are created using default values, and if you plot more than one mathematical function you may want to alter these. In the following example you change the y-axis title:

```
> curve(sin, -pi*2, pi*2, lty = 2, lwd = 1.5, ylab = 'Function', ylim = c(-1,1.5))
> curve(cos, -pi*2, pi*2, lty = 3, lwd = 1, add = TRUE)
> legend(x = 'topright', legend = c('Sine', 'Cosine'), lty = c(2, 3),
lwd = c(1.5, 1), bty = 'n')
> title(main = 'Sine and Cosine functions')
```

In this case you alter the y-axis title, and also the limits of the axis to accommodate the legend. You make the first curve in one style (lty = 2) and a bit wider than "normal" (lwd = 1.5). The second curve is set to a different style (lty = 3) and, of course, set to the same axis limits as the sine curve (from – * 2 to * 2). Next, you add a legend to differentiate between the two curves; note that you specify the legend text and the line styles and thicknesses explicitly. The final graph looks like Figure11-14.

Figure 11-14:



If you require a more complex mathematical function, you must specify it separately using the function() command; then you can refer to your function in the curve() command. In the following example you create a simple mathematicalfunction:

```
> pn = function(x) x + x^2
> pn
function(x) x + x^2
```

The function() command has two parts: the first part is a list of arguments and the second part is the list of commands that use these arguments. In this case, the single argument is in the parentheses, x; the value of *x* is added to *x2* and this is what the function does, creates *x + x2* values. You can see how it works by using the new function on some values directly likeso:

```
> pn(1)
[1] 2
> pn(2)
[1] 6
> pn(1:4)
[1]      2     6 1220
```

When you use the new function as part of a curve() command, you therefore plot the following equation:

$y = x + x^2$

Recall the polynomial model you had before:

```
> bbel.lm

Call:
lm(formula = abund ~ light + I(light^2), data = bbel)

Coefficients:
(Intercept)                 light         I(light^2)
     -2.00485              2.06010           -0.04029
```

You could set a function to mimic this formula and then plot that like so:

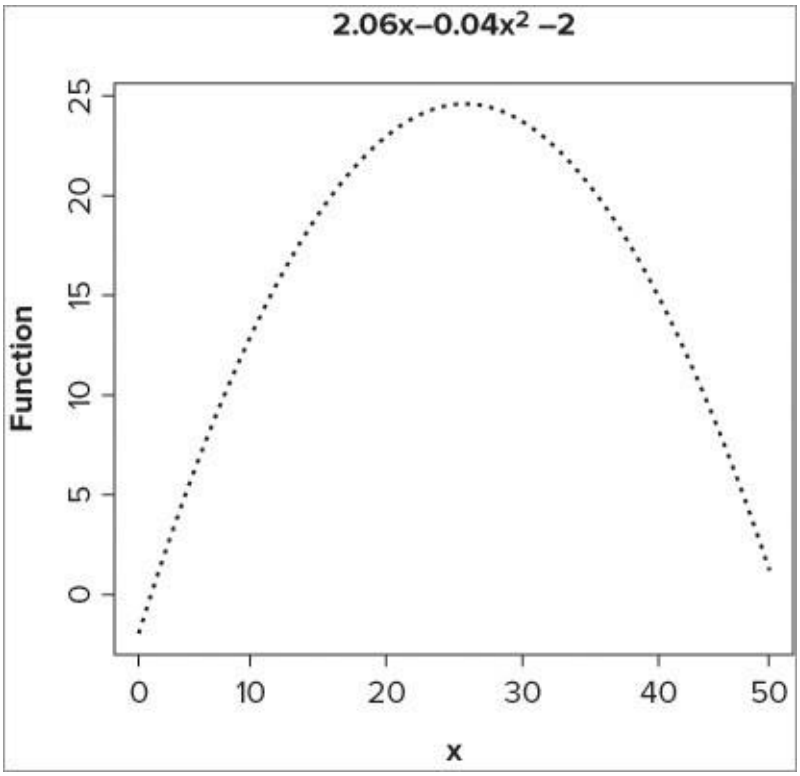
```
> pn = function(x) (2.06*x)+(-0.04 * x^2)-2
> curve(pn, from = 0, to = 50, lwd = 2, lty = 3, ylab = 'Function')
> title(main = expression(2.06*x~-0.04*x^2*~-2))
```

The first line sets the polynomial function; for every value of *x* you type, the function will evaluate $y = 2.06x - 0.04x^2 - 2$. Now you use the curve() command to draw this function; you set the line type to dashed and make it a little wider than normal. In the last line you set an expression() to create the formula as the title. The graph that results looks like Figure 11-15.

Table 11-11: Instructions Used with the curve() Command

Instructio	Explanation
expr	Amathematicalexpressiontoevaluate;forexample,log,sin,cos.Apredefinedfunctio n()
from =	The starting value for the x-axis and expression to be evaluated.
to =	The ending value for the x-axis and expression to be evaluated.
n = 101	The number of points to evaluate.
lty = n	The line type; for example, 2 = dashed, 3 = dotted.
lwd = n	The line width. 1 = “normal”, > 1 makes line wider, < 1 makes it thinner.
col = ‘color’	The line color, usually as a text string. For example, “gray50”.
type = ‘l’	Thetypeofplotused.‘l’isthedefaultanddrawsalineonly,‘b’producessectionsofline with pointsbetween.
pch = n	The plotting character, usually as a numeric value but can be a text string. For (the same as pch = 3). Used only if lty = ‘b’, ‘p’, or ‘o’.
add =	If set to TRUE, the curve is added to an existing plot.

Figure 11-15:



Adding Short Segments of Line to an Existing Plot

You can add short segments of line to an existing plot using the segments() command; you saw this before when adding error bars to a bar chart. The basic form of the command is as follows:

segments(x0, y0, x1, y1, ...)

You provide the starting coordinates and the ending coordinates. You canalso use a host of other graphical instructions to alter the appearance of these segments; for example, lty, col, and lwd to alter line type, color, and width, respectively.

If you have a series of coordinates you can use these to draw shapes onto existing plots.

Adding Arrows to an Existing Graph

You can use the arrows() command in much the same way as the segments() command. The arrows() command draws sections of line from one point to another and adds arrowheads as specified in the command instructions. The basic form of the command is asfollows:

arrows(x0, y0, x1, y1, length = 0.25, angle = 30, code = 2, ...)

The command requires the x and y coordinates of the starting and ending points. The length = instruction specifies the length of the arrowhead (if appropriate) and the angle = instruction is the angle of the head to the line; this defaults to 30˚. The code = instruction specifies on which end(s) the arrowheads should be drawn; 1 = the back end (that is, *x0*, *y0*), 2 = the front end (that is, *x1*, *y1*), and 3 = bothends.

One use for the arrows() command is to create error bars with hats. Previously you used

the segments() command to add error bars but you used three separate commands, one for the bars and one for each of the hats. Youcan use the arrows() command to add the hat sat either end:

```
> arrows(bp, bf.m+bf.se, bp, bf.m-bf.se, length = 0.1, angle = 90, code = 3)
```

In this case you use a length of 0.1 to give a short hat and set the angle at 90˚. The code = 3 instruction sets heads at both ends. The values for the coordinates are determined from the data; look back at the previous section on error bars for a reminder on how to dothis.

You can specify other graphical instructions to alter line style, color, and width, for example. In the following example you see some of these instructions applied:

```
> 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


> plot(count ~ speed, data = fw, pch = '.')
> s = seq(length(fw$speed)-1)
> s
[1] 1 2 3 4 5 6 7

> arrows(fw$speed[s], fw$count[s], fw$speed[s+1], fw$count[s+1],
length = 0.15,
angle = 20, lwd = 2, col = 'gray50')
```

You complete the following steps to achieve the preceding example:

1. You have a simple data frame and use the plot() command to create a scatter graph, setting the plotting character to a period(.).

2. Next, you create a simple sequence that is one shorter than the length of your data; you had eight rows so you need to end up with sevenvalues.

3. Next, you use the arrows() command to add arrows that go from point to point; you set the head length to 0.15 and the angle to 20˚. In this case the default is for the arrowheads to appear at the far end, so you do not need to specify code =2.

Matrix Plots (Multiple Series on One Graph)

Sometimes you need to produce a plot that contains several series of data. If your data are categorical, the barplot() command is the natural choice (although the dotchart() command is a good alternative). If your data are continuous variables, a scatter plot of some kind is required. You saw previously how to add extra points or lines to a scatter plot. However, the matplot() command makes a useful and powerfulalternative.

In Chapter 8, “Formula Notation and Complex Statistics” you looked at interaction plots using

the interaction.plot() command. You can produce a similar graph using the matplot() command, which plots the columns of one matrix against the columns of another. This command is particularly useful for plotting multiple series of data on the same chart. The general form of the command is as follows:

matplot(x, y, type = 'p', lty = 1:5, pch = NULL, col = 1:6)

The x refers to the matrix containing the columns to be used as the x-data. The y refers to the matrix containing the columns to be plotted as the y-data. The two matrix objects do not have to contain the same number of columns, but they do need to have the same number of rows. By default, only points are plotted, but if you select type = 'b', for example, the style of line differs for each column in your y-matrix; the lty = instruction uses values from one to five and recycles these values as required. If you require different line types, you can specify them yourself. The plotting characters are set by default to use numbers from one to nine, then zero, then lowercase letters, and finally the uppercase letters. You can, of course, set the symbols you require using the pch = instruction explicitly. You can also set the colors of the plotted lines/characters by altering the col = instruction; by default this uses colors one through six (black, red, green, blue, cyan,violet).

The following example shows two data matrices. The first contains two columns and these relate to two response variables. The second matrix is a single column representing a predictor variable:

```
> ivert
      sfly mfly
 [1,]   26    4
 [2,]   23    5
 [3,]   33   12
 [4,]    6    9
 [5,]    3   15
 [6,]    4   10
 [7,]   20    8
 [8,]    2   22
> spd
      speed
 [1,]     3
 [2,]     4
 [3,]     4
 [4,]     5
 [5,]     6
 [6,]     7
 [7,]     7
 [8,]     9
```

To create a basic plot, you can try the following:

```
> matplot(spd, ivert)
```

In this case you have a single *x*-variable (spd), and the two columns in your y- data are plotted against this. Using all the defaults, you end up with a plot that shows two sets of points, with the plotting characters being black "1" and red "2".

In the following example you add some additional instructions to produce a more customized

plot and follow this up with alegend:

```
> matplot(spd, ivert, type = 'b', pch = 1:2, col = 1, lty = 2:3, xlab = 'Speed',
ylab = 'Invertebrate count')
> legend(x = 'topright', legend = c('Stonefly', 'Mayfly'), pch = 1:2, col = 1,
bty = 'n', lty = 2:3)
```

Your matplot() command produces a plot with both lines and points using the type = 'b' instruction. This time you specify the plotting characters explicitly using pch = 1:2 (an open circle and an open triangle). You set both lines to black using col = 1, but vary the line type using lty = 2:3 (producing a dashed line and a dotted line). Finally, you add explicit axis titles using the xlab
=and ylab =instructions.

Whenyouaddyourlegendusingthelegend()command,youcancopysome of the instructions because you need to match the line type, color, and plotting characters.Todosoperformthefollowingsteps:

1. Beginbysettingthelegendpositionatthetoprightoftheplotwindow.
2. Next,specifythetextforthelegendexplicitly.Therestoftheinstructions match the matplot() ones except for bty = 'n', which suppresses the borderaroundthelegend.
3. ThefinalplotlookslikeFigure11-17.

If you were to have two columns in your x-values matrix, the first column of the x-matrix would match up to the first column of the y-matrix, and so on. If you have more columns in the y-matrix than in the x-matrix, the columns ofthe x-matrixarerecycledasnecessary.Inthisway,youcanarrangeyourdatasothat youproducethemulti-seriesplotyouneed.

Figure 11-17:

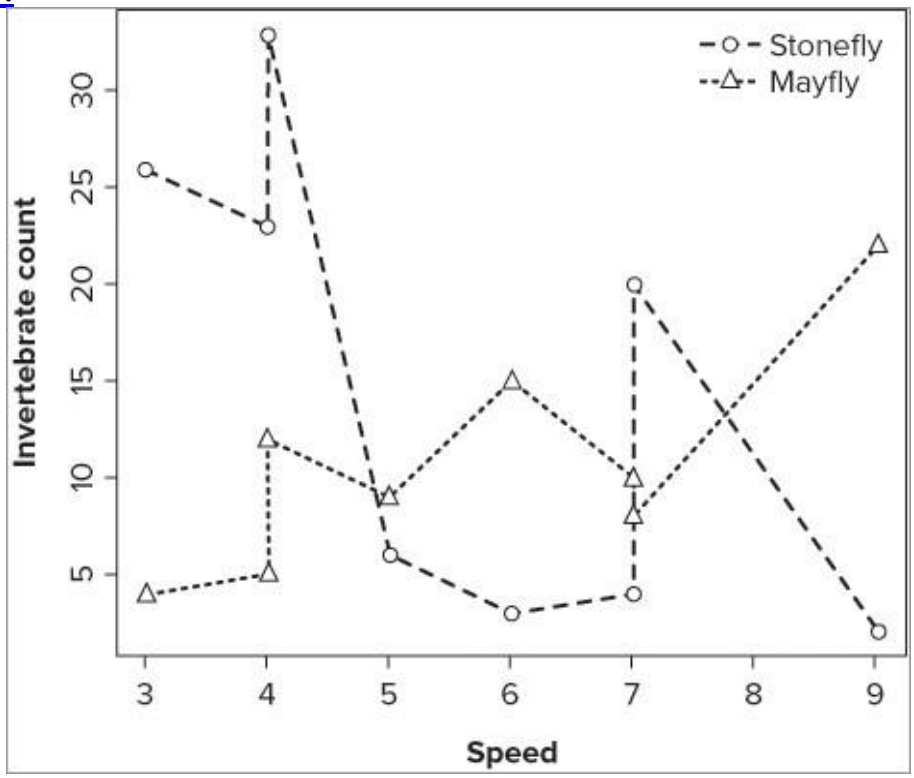


Table 11-13: Options for Matrix Plots

Instruction	Explanation
x	A matrix of numerical data, the columns of which will form the x-axis of
y	Amatrixofnumericaldata,thecolumnsofwhichwillformthey-axisoftheplot.Ifonlyone matrixisspecified,thiswillbecomethey-axisdataandasimpleindexwillbeusedforthex- axis.
type =	Thetypeofplot.Defaultsto'p'formatplot()andmatpoints();defaultsto'l'for matlines().Allowabletypesare'n','p','b',and'o'fornoplot,points,both,and overplot(similartobothexceptpointsoverlietheline).
lty = 1:5	Linetype.1=plain,2=dashed,3=dotted.Thedefaultvaluesare1:5,andifmoreare requiredthevaluesarerecycled.
lwd = 1	Line width. 1 = normal, > 1 makes line wider, < 1 makes line narrower.
pch = null	Theplottingsymbolsused;bydefaultnumbers1–9areused,followedby0,thenlowercase and uppercase letters. Explicit symbols may be specified by numerical value or as a text
col = 1:6	Color of the plotted line and/or points. The default values are 1:6 (black, red, green, blue, cyan,orviolet),whicharerecycledifrequired.Colorsmaybespecifiedasanumerica
….	Other graphical instructions. For example, ylim, xlab, or cex.

Multiple Plots in OneWindow

It is sometimes useful to create several graphs in one window. This could be because they are closely related and you want to display them together. You could make separate plots and then later position them together using a graphics program or your word processor. However, it is more efficient to split a graphical window into sections and draw your graphs into the sections directly from R. You are able to achieve this in a couple of ways, as you see in the followingsections.

Splitting the Plot Window into EqualSections

You can split the graphical window into several parts using the graphical instructionsmfrow()andmfcol();youcanaccesstheseinstructionsonlyviathe par()command.Usually,itisagoodideatostorethecurrentpar()instructions so that they can be recalled/reset later. The mfrow() and mfcol() instructions bothrequiretwovalues,thenumberofrowsandthenumberofcolumns,likeso:

mfrow = c(nrows, ncols) mfcol
= c(nrows, ncols)

Once set, the split plot window remains in force until altered or reset to its original values. In the following example you split the window into four; that is, two rows and two columns:

```
> opt = par(mfrow = c(2,2))
> plot(Length ~ BOD, data = mf, main = 'plot 1')
```

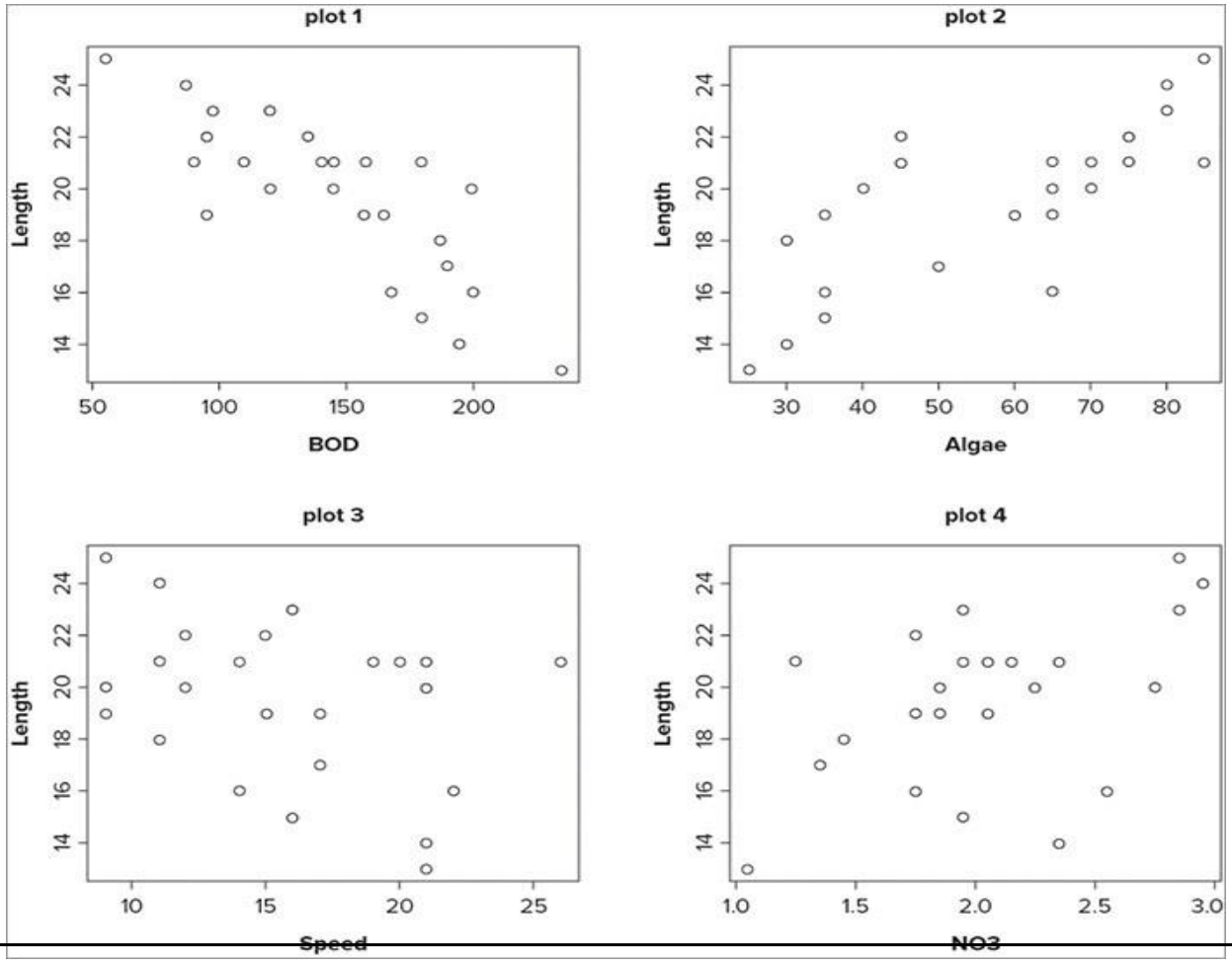
```
> plot(Length ~ Algae, data = mf, main = 'plot 2')
> plot(Length ~ Speed, data = mf, main = 'plot 3')
> plot(Length ~ NO3, data = mf, main = 'plot 4')
> par(opt)
```

The first command sets the number of rows and columns for the plot window; note that you create an object to hold the current settings. The next four lines produce simple scatter plots; this is where the difference between mfrow() and mfcol() becomes evident. In the case of mfrow() the plots are created row by row, whereas if you had set mfcol() the plots would fill up column by column. At the end you reset the window by calling up the object you created earlier. The final plot looks like Figure11-18.

You can skip a plot by using the plot.new() command. The result of issuing a plot.new() command is that the current plot is finished; if you have a split window, you skip over the next position in the sequence. You must remember that the current plot is not the one that you just drew, but the one that is ready to draw into! In the following example you set the window to four sections again, but draw plots bycolumn:

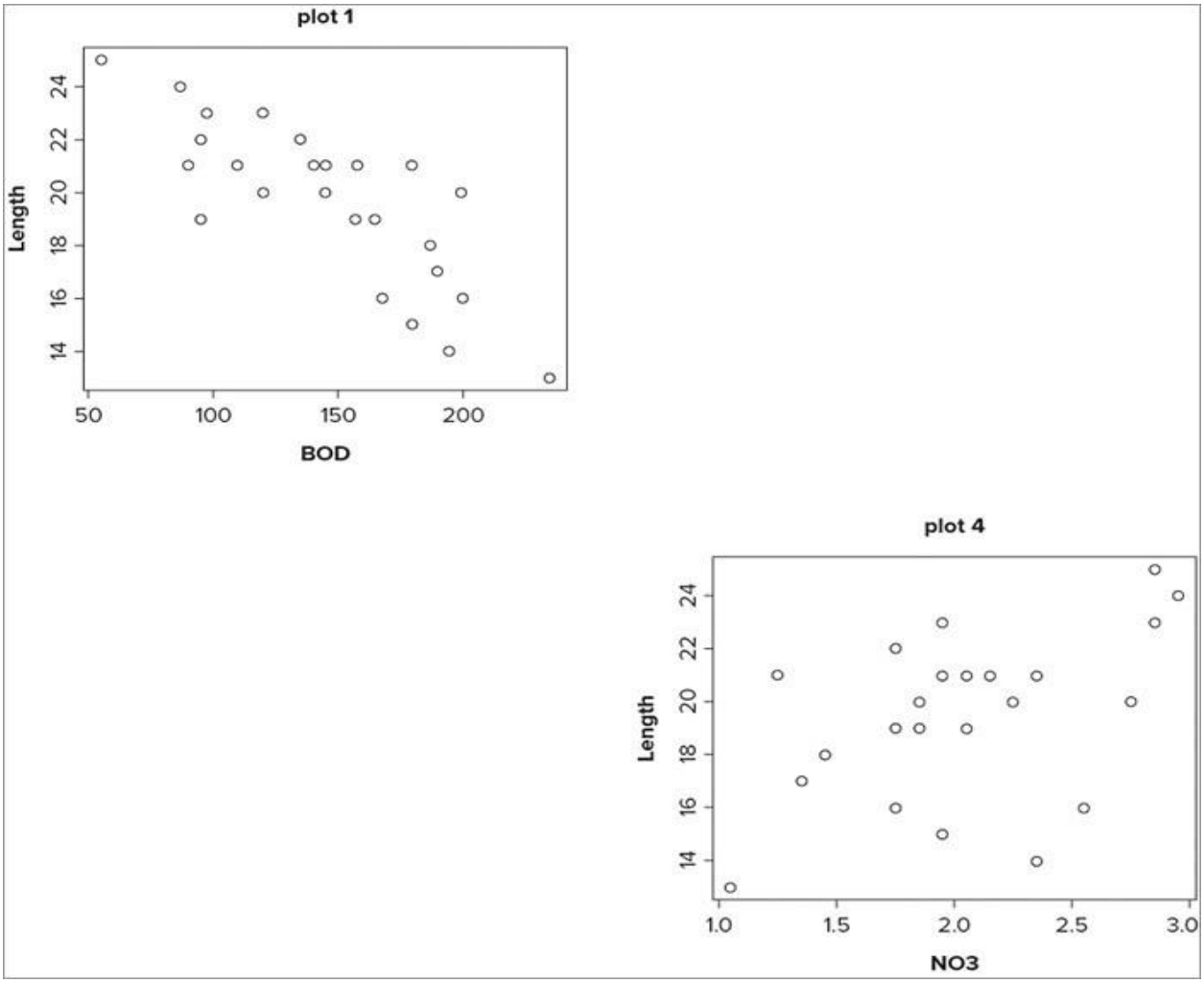
```
> opt = par(mfcol = c(2,2))
> plot(Length ~ BOD, data = mf, main = 'plot 1')
> plot.new()
> plot.new()
> plot(Length ~ NO3, data = mf, main = 'plot 4')
> par(opt)
```

Figure 11-18:



After you issue the mfcol() instruction the plot is ready (although you cannot see it), and so the first plot will go into the top-left position. The next plot is ready to go into the bottom left, but you use plot.new() to skip it and get the next ready instead. However, you decide to skip this too and so the next plot goes into the bottom section of the window. The final graph appears like Figure 11-19. The blank areas might be useful to add text, for example, or perhaps some othergraphic.

Figure 11-19:



You can draw directly into a portion of the plot window by using the `mfg` instruction with the `par()` command. The plot window must already have been set via the `mfrow` or `mfcol` instruction. You specify `par(mfg = c(i, j))` where `i` and `j` are the row and column coordinates to be used.

Splitting the Plot Window into Unequal Sections

You can control the sections of the plotting window more exactly using the split.screen() command. This command provides finer control of the splitting process and you can also place a plot in exactly the section you require. The basic form of the command is asfollows:

split.screen(figs = c(rows, cols))

This is similar to the mfrow() instruction you saw in the previous section, but you can go further; you can subdivide each of the sections you just created as well.Thefollowingexampleisagoodwaytoillustratethepossibilities:

> split.screen(figs = c(2, 1)) [1] 1 2
> screen() [1]
1

In this case you divide the graphics window into two rows and one column; the command gives you a message showing that you now have two areas available. The second command, screen(), checks to see which is the current window;yougeta1inthisinstance,whichindicatesthatyouareatthetop.You switchtothebottomrow(screentwo)becauseyouwanttosubdivideit:

> screen(2)
> split.screen(figs = c(1, 2)) [1] 3 4

You start by simply switching to screen number two using screen(2), then you split this into more parts; in this case you decide to make a single row and two columns. Your graphics window is now split into four parts; this is not immediately obvious! You have the original split, which was the top half and the bottom half. You have the bottom half also split into two parts; this makes four altogether. You can see how many parts you have by using the close.screen() command likeso:

> close.screen() [1] 1
2 3 4

When you use this command without any instructions, you get a list of the available screens. You can plot into any of the screens by selecting one and then using the plot() command (or any other command that produces a graph). In the following example you make a scatter plot in screens two and one:

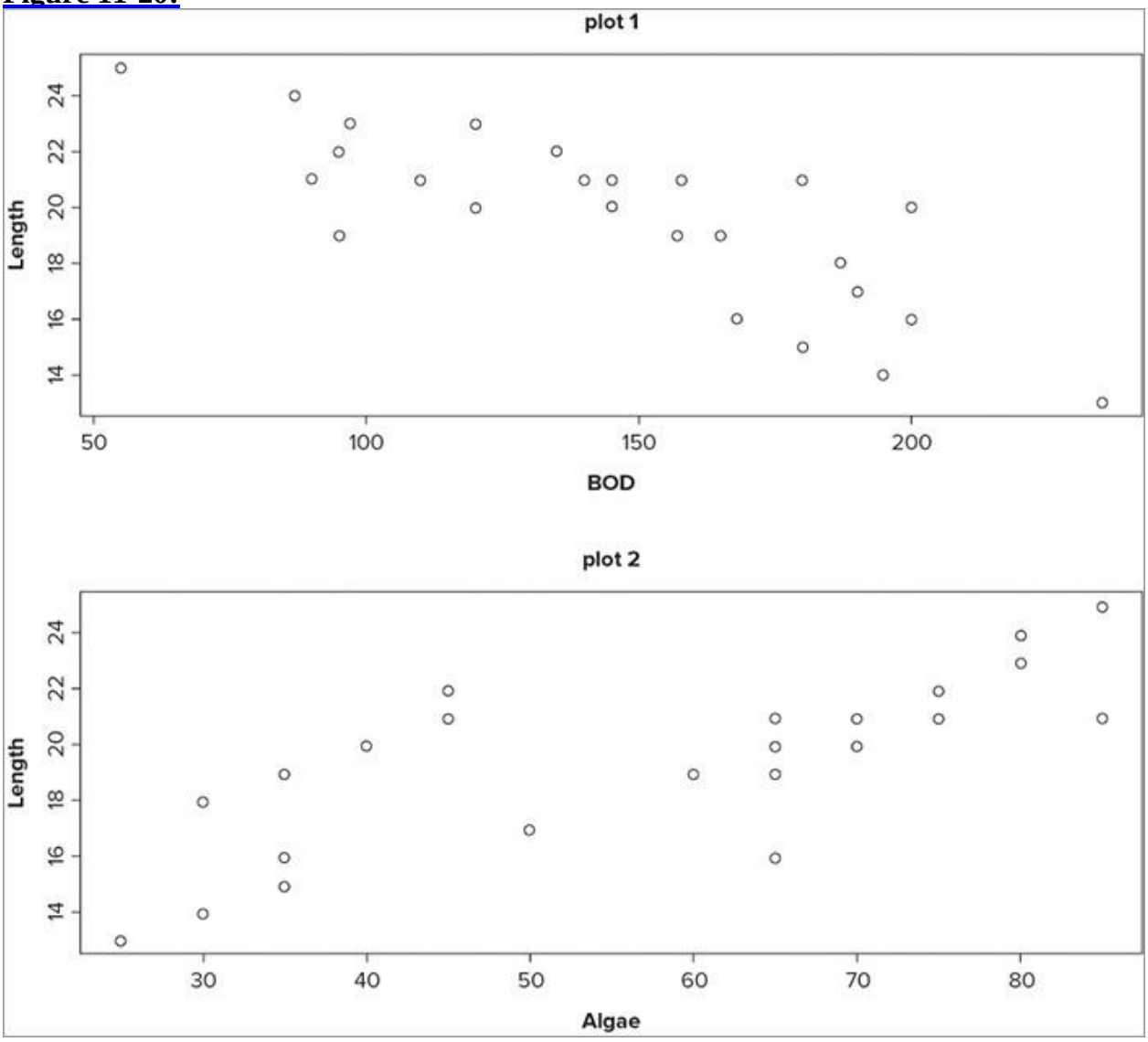
> screen(2)
> plot(Length ~ Algae, data = mf, main = 'plot 2')
> screen(1)
> plot(Length ~ BOD, data = mf, main = 'plot 1')

The graph that results looks like Figure 11-20.

You still have split the bottom part into two sections (that is, one row and two columns), so you could create a more complex plot. You begin by erasing the bottom plot (screen two) using the erase.screen() command like so:

> opt = par(bg = 'white')
> erase.screen(n = 2)
> par(opt)

Figure 11-20:



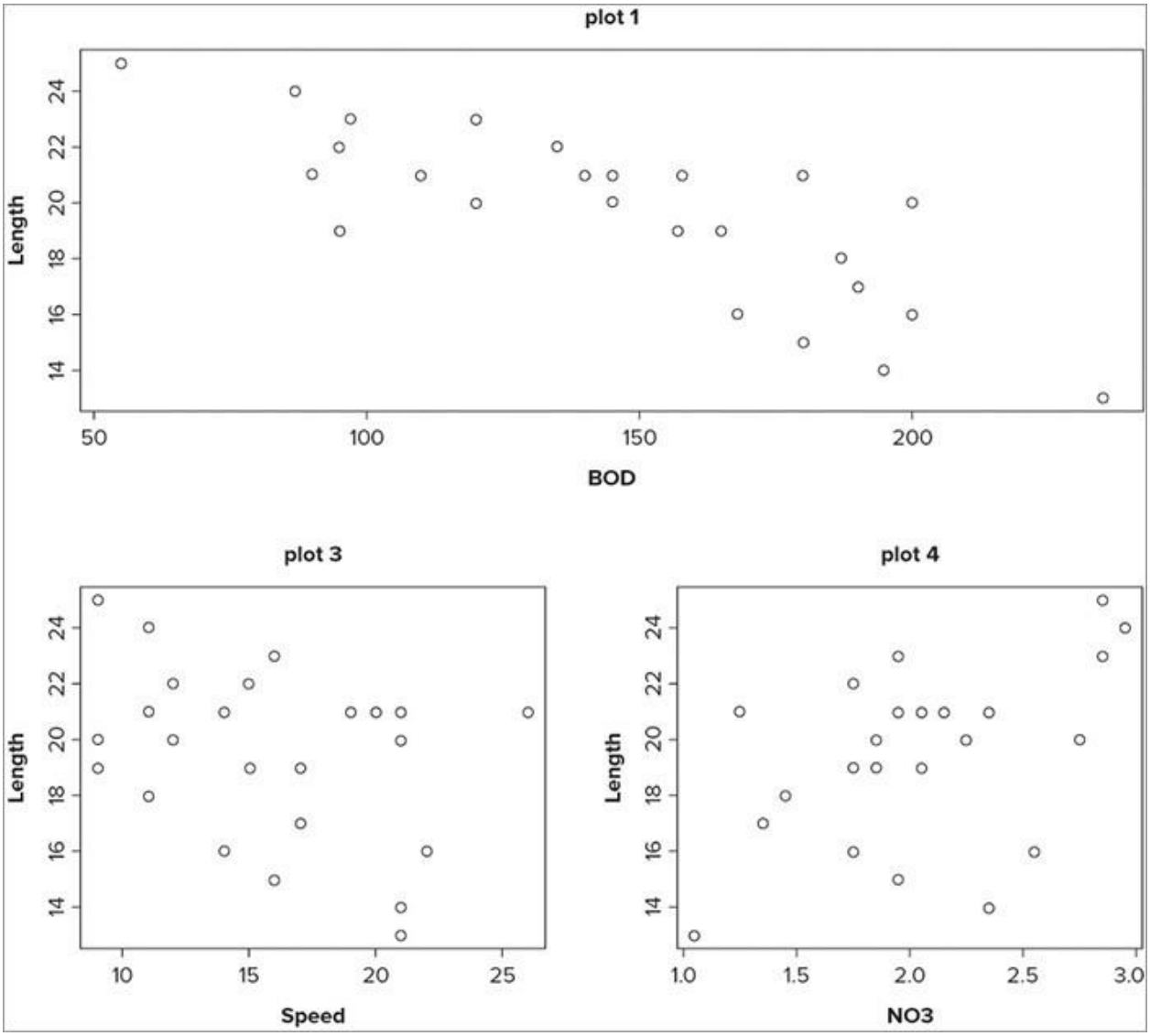
The default for this command is to use the currently selected screen, so you must be careful to specify which screen you want to erase. By default, the background color of the plot is used to erase the drawing; in many cases this is transparent, so you set it to white in this case to ensure that you wipe out the current plot. You can now draw into the cleared area at the bottom of the plot window:

```
> screen(3)
> plot(Length ~ Speed, data = mf, main = 'plot 3')
> screen(4)
> plot(Length ~ NO3, data = mf, main = 'plot 4')
```

Recall that you split the window into four parts with screens three and four making up the bottom half. You start by selecting screen(3) and plotting a scatter plot; you then select screen(4) and make another plot. The resultis

shown in Figure 11-21.

Figure 11-21:



You can use the close.screen()command to remove splits:

> close.screen(n = 3:4) [1] 1 2

Here you see that you retain screens one and two. If you want to close all the screens,youcanspecifythisasaninstructionintheclose.screen()command likeso:

> close.screen(all.screens = TRUE)

> close.screen() [1]

FALSE

When you use the close.screen() command without any instructions, you

nowseeFALSEasaresult,indicatingthatyouhavenoscreensremaining(thatis, nosplits,onlythebasicgraphicalwindow).

When you use split screens it is advisable to complete an entire plot before switching to a new screen; so complete the addition of titles, extra text, lines, and points before moving to the next. Table 11-14 shows a summary of splitting screens.

Table 11-14: Summary of Screen (Graphics Window) Splitting Commands and Options

Command/Instructi	Explanation
split.screen(figs, screen)	Splits the graphical window into subsections.
figs	Thenumberofrowsandcolumnsrequired.Forexample,c(2,2)setstworo ws and twocolumns.
screen	Anumericalvaluespecifyingwhichscreenistobesplit;onlyworksifthe graphicalwindowisalreadysplit.
screen(n =)	Switchestothescreenspecifiedbyn.Ifnismissing,thecurrentscreennumbe ris given.
erase.screen(n =)	Erasestheselectedscreenbydrawingoveritinthecurrentbackgroundcol or; thismaybetranslucentandnotappeartohaveanyeffect.
opt = par(bg = 'white') par(opt)	Sets the background color to white, which enables a screen to be erased. The commandalsostoresallgraphicalparameters soallcanberesetusingpar(op
close.screen()	Returns the numbers of the available screens remaining.
n	Selectsthescreentobeclosed;thenumbersoftheremainingavailablescreen sis returned.
all.screens = TRUE	If set to TRUE, all screens are closed and the single graphical window

Exporting Graphs

Once you have made a graph, you will have it on your computer screen. This may well be extremely useful as a diagnostic tool, but to reach a wider audience you will need to transfer your graph to another location. If you are going to use the graph in a report or presentation, you will need to place it into an appropriate program. Previously (in Chapter 7) you saw how to use copy and paste to transfer graphs directly into other programs and how to save the graphic window as a file on disk. Here you see a brief review of these processes before looking at ways to fine-tune your graphics via the device driver, which can produce high- quality graphicsfiles.

Using Copy and Paste to Move aGraph

For many purposes, using the clipboard is simple and produces acceptable quality graphics for your word processor or presentation. You can resize the graphic window using your mouse like you would for any other program windowandyourgraphwillberesizedaccordingly.

If you want more control over the size of the graphics window, you can create a blank window with the dimensions you require using one of the following commands:

windows(height, width, bg)
quartz(height, width, bg) X11(height,
width, bg)

Which command you need depends on the operating system you are using at the time.

- For Windows users, the windows() command creates a new graphic window.
- ForMacintoshusers,thequartz()commandopensagraphicwindow.The X11()commandalsoopensagraphicwindow,butintheX11application.
- ForLinuxusers,theX11()commandopensagraphicwindow.

You set the height and width in inches; the defaults depend on your system. You can also specify the background color using the bg instruction; the default is bg =“transparent”.

The default size for the graphics window can be set in the options for the GUI in Windows and Macintosh. In Linux you have to set a default in a profile file to run each time R loads. For most users the defaults are acceptable (7 inches), and for special uses it is quite simple to use the X11() command to produce a custom-sizedwindow.

Saving a Graph to a File

Saving to a file generally makes a better quality graphic than using the clipboard. As you saw in Chapter 7, you have a variety of options for saving graphics files according to your operating system.

Windows

When you have a graphics window you will see that it contains a menu bar. The File menu enables you to save the contents of the window to a disk file. You have several choices of format, including PNG, JPEG, BMP, TIFF, and PDF. The JPEG option also allows you to specify the compression.

Macintosh

The Mac does not present you with a menu as part of the graphics window. Once the graph is selected, the File menu enables you to save your graph as a file. The default is to use PDF format. It is not trivial to alter the default. If you need a file in a different format, you can use the device driver, as you see shortly.

Linux

In Linux there is no GUI, and R runs via the Terminal application. This means that you cannot save the graphics window in this fashion; you must use the device driver in somemanner.

Using the Device Driver to Save a Graph to Disk

The device driver allows more subtle control of your graphics and the quality of the finished article. If you are a Linux user, you have to use the device driver in some way unless a simple copy-and-paste operation will suffice. In Chapter 5 you saw how to use the device driver briefly. Here you see a few more of the options available toyou.

You can think of the device driver as a way to send your graphics to an appropriate location; this may be the screen, a PNG file, or a PDF file. You can use the device driver in two mainways:

- Send an existing screen graphic to a file. Create
- a graphic file directly on disk.

The device driver can accept a variety of instructions, including the size of the graphic as well as the color of the background and the resolution (DPI). Here you see the device driver in action for saving PNG and PDFfiles.

PNG Device Driver

You access the PNG device via the png() command, which has the following general form:

```
png(filename = "Rplot%03d.tif", width = 480, height = 480, units = "px", bg="white", res
=NA)
```

You must specify a filename in quotes; the default will be used if you do not. This file will be written to your default directory. To alter the location you must either alter the default or specify the location explicitly as part of the filename. The height and width are specified in pixels by default, and the units instruction controls which units are used; you can specify “in”, “cm”, or “mm” as alternatives to the “px” default. The bg instruction sets the background color, and the res instruction controls theresolution.

The resolution is not recorded in the file unless you specify it explicitly. Many graphics programs will assume that 72 dpi has been used, so in practice it is a good idea to specify one—72 dpi is the standard for screen use, and with the 480 pixels size this comes out to 6.67 inches. You can use the png() command to create a file directly, or to copy a screen graphic todisk.

PDF Device Driver

The PDF device driver is accessed via the pdf() command and has similar options to the png()command.

```
pdf(file = "Rplot%03d.pdf", width, height, bg, colormodel)
```

The filename must be given in quotes and are saved to your default directory unless you alter it or specify the location explicitly as part of the filename. The height and width are measured in inches and use the default of 7 inches. The background color is set to “transparent”by default.

The colormodel instruction enables you to specify the general color of the plot;thedefaultis“rgb”,whichproducescoloredgraphics.Youcanalsospecify “gray”toproduceagrayscaleplot.

Copying a Graph from Screen to Disk File

These are very similar, and you can treat them the same for most purposes. The device instruction is essentially the png() or pdf() command that you saw previously (you can also use tiff(), jpeg(), and bmp()commands).

The difference between the two dev commands is that dev.print() writes the graphic file immediately, whereas the dev.copy() command opens the graphic file and sends a copy but does not finish. This means that you can add additional commands to the graphic on disk without altering the one on the screen! Once you are finished with the graphic, you must close the device to finish writing the file and complete the operation. You do this using a newcommand:

```
dev.off()
```

No additional instructions are required. The graphics file created is closed and becomes

available as soon as the dev.off() command is executed. The dev.print() command does not need you to do this because the file is written and closed in onego.

Making a New Graph Directly to a Disk File

1. Create the device using the appropriate driver. You need to specify the filename and the size, as well as the resolution and any special background color.

2. Issue the graphics commands that you need to produce the basic graphic. This involves the plot() command or something similar (for example, boxplot(), barplot(), or dotchart()commands).

3. Addadditionalgraphicscommandstoembellishyourgraphsuchas title() to add axis titles or abline() to add a line of best-fit.

4. Finish the plot by closing the graphics device using the dev.off() command.

Your computer will be set up to create graphics windows of a certain size. When you make a graph directly on disk, it is important to match the resolution you require to the appropriate size (in pixels); otherwise, the text, plotting characters, and so on will either be far too small or far too large. In the following activity you create a graph and explore the effects of altering the resolution.

Try It Out: Save a Graph to Disk and Explore the Effects of Resolution

For best results you should make a trial plot on the screen. Start by making a graphics window of a set size using the windows(), quartz(), or X11() commands as appropriate. Then create your graph. Once you have a graph that you are happy with, you can match the size and resolution by specifying the dimensions as *width * dpi* and *height * dpi*. So, if you get a good result with a 7 inch x 7 inch window and require a dpi of 300, you can specify 7 * 300 as the size. You can recall the graphics commands that you used to create your graph simply using the uparrow.

Use the fwidata object from the Beginning.RDatafile for this activity.

1. Make a graphic on the screen to check the layout using your default settings:

```
> plot(sfly ~ speed, data = fwi, main = 'Scatter plot', pch = 16, cex = 2,  las
= 1)
> abline(h = mean(fwi$sfly), lty = 3, lwd =2)
> abline(v = mean(fwi$speed), lty = 3, lwd =2)
> abline(lm(sfly ~ speed, data = fwi), lty = 2, col ='blue')
> text(max(fwi$speed), mean(fwi$sfly)+ 0.5, 'Mean sfly', pos = 2, font =  3)
> text(mean(fwi$speed), max(fwi$sfly), pos = 4, srt = 270, 'Mean speed', font = 3)
```

2. Now create a new graphics window at a fixed size using the appropriatecommandforyourOS:

```
> windows(width = 7, height = 7)
> quartz(width = 7, height = 7)
> X11(width = 7, height = 7)
```

3. Redrawthegraphinthenewwindow.

4. Now create a PNG file using the device driver. Set the resolution to 300dpi:

```
> png(file = '7in 300dpi.tif', height = 2100, width = 2100, res = 300, bg = 'white')
```

5. Send the graphics commands once again, and then finish by closing the devicedriver:

```
> Graphics commands here..
> def.off()
```

6. CreateanewPNGfileandsettheresolutionto150dpi:

```
> png(file = '7in 150dpi.tif', height = 2100, width = 2100, res = 150, bg = 'white')
```

7. Send the graphics commands once again, and then finish by closing the devicedriver:

```
> Graphics commands here..
> def.off()
```

8. Create a new PNG file and set the resolution to 600dpi:

```
> png(file = '7in 600dpi.tif', height = 2100, width = 2100, res = 600, bg = 'white')
```

9. Send the graphics commands once again, and then finish by closing the devicedriver:

```
> Graphics commands here..
> def.off()
```

10. Go to your working directory and look at the differences in the graphics (using your OS).

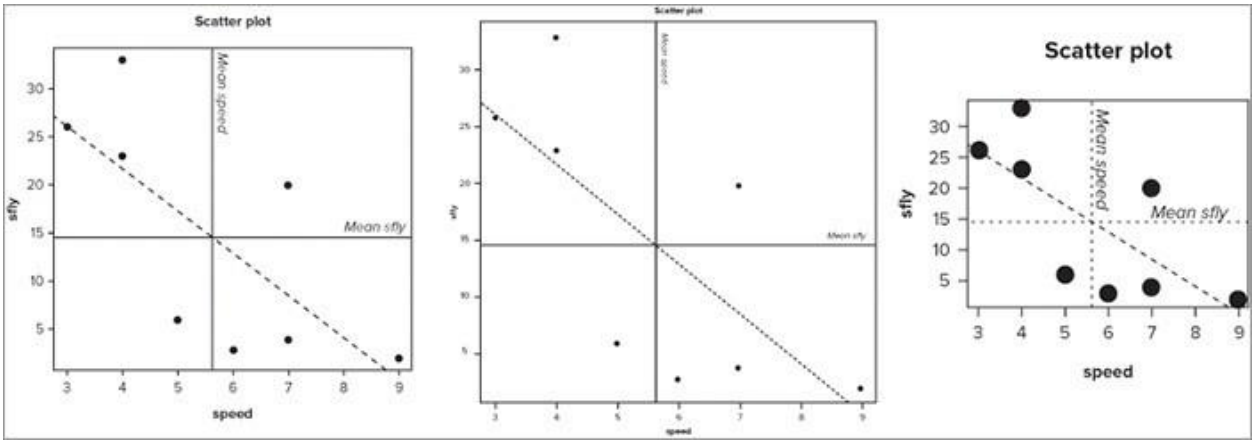
How It Works

The basic graphics commands make the plot using some embellishments that you have already seen in the text. The appearance of the graph will be slightly different depending on the default size of your graphics window. Therefore, to make sure you can visualize the final output, you create a new window using a set size; here you used seven inches for both width andheight.

The device driver requires size and resolution information. In the first caseyouused2100pixelsand300dpi.Inotherwords,youused7×300 = 2100 pixels as the size to match the onscreen graphic. Once the device is ready, you send the graphics commands to actually draw the plot and close using the dev.off()command.

The next two graphs you produce keep the same number of pixels but alter the resolution. In Figure 11-22 you can see the effect this has on the way the graphic appears. Low resolution and large size make everything very small; as you increase the resolution, everything becomeslarger.

Figure 11-22:



Summary

- You can add error bars to graphs using segments() or arrows() commands. You can add
- legends using the legend instruction from some graph commands; for example, barplot() or separately using the legend() command.
- You can define color palettes using the palette()command.
- Use the expression() command to create complex text including superscripted and subscripted typeface. It can also create mathematical text. You can alter the plot margins to accommodate text by using the mar instruction within the par()command.
- Text labels on axes can be altered in various ways; for example, altering the orientation, color, size, and font.
- You can add text and points to existing graphs using the text()and points()commands.
- Various commands can add lines to graphs, including abline(), lines(), and curve(). The spline() command can bend sections of straight line to create a smooth curve.
- The matplot()command can plot multiple series on one chart. The matpoints()and matlines()can add to an existing plot.
- The graphical window can be split into sections to allow multiple graphs to be produced in one window. The sections do not have to be the same size. The device driver can
- create a blank graphics window as well as copy an existing graphic to a disk file. You can also create a graphic directly as a diskfile.

Chapter 12

Writing Your Own Scripts: Beginning to Program

What You Will Learn In This Chapter:

- How to store series of commands as snippets to be used with copy/paste How to make
- your own help file
- How to create simple customized functions
- How to edit, store, and recall customized functions How to add
- notes/annotations to your scripts
- How to create complex program code

Because R is a programming language, you have great flexibility in the approach you can take to running it. When you first begin to use R you will probably type commands directly from the keyboard. Later, as you become more confident, you will likely use snippets of commands stored in other areas, like a text file. The next step is to create simple functions that carry out something useful; you can call up these functions time and time again, and can save a lot of typing and effort. As your confidence and ability grow you will move on to creating larger *scripts*, that is, sets of R commands stored in a file that you can execute at anytime.

Scripts can be especially useful because they enable you to prepare complex or repetitive tasks, which you can bring into operation at any time. Indeed, R is built along these lines, and you can think of the program as a bundle of scripts; by making your own you are simply increasing the usefulness of R and bending it to meet your own specificrequirements.

Programming R is a wide subject in its own right. This chapter introduces you to the basic ideas so that you can set off on your own journey of discovery. Of course, you have gained a lot of experience at using R up to this point, so the step up to creating your own programs is only a smallone.

Copy and Paste Scripts

R is very flexible, and because it accepts plain text to drive the commands, you can store useful snippets to use at a later date. You can copy and paste the text from a word processor (or other program) into R and either run the command "as is" or edit the command before you press Enter.

Make Your Own Help File as PlainText

As you learn how to use R it is a good idea to keep a plain text file as a "notepad" (on Windows computers the Notepad.exe program is a goodchoice for this task). You can use this text file to keep notes and examples of R commands. However, a plain list of commands without explanation is not helpful. You can,ofcourse,addexplanatorynotesinsomewayasyougoalong; thefollowingexamplemightbepartofyournotesfile:

Code to work out means of columns in a data frame:

```
---------------------------------
apply(data, 2, mean, na.rm = TRUE)
---------------------------------
```

data = name of data frame
2 = columns (1 for rows)
mean = the mean command (can use others) na.rm = TRUE =
remove NA items ifappropriate

To use this you can simply copy the text to the clipboard and paste it into the R console window; in the example here you want the line of text that is between the dashed lines. You can edit the name of the data and add a name for the result as you like. This is a good way to build up a library of commands that you can use and become familiar with.

When you copy command lines from R, you inevitably copy the >character that forms the "command entry point" as well. This book has used this approach so that you can see which lines were typed from the keyboard and which lines are results (that is, generated by R itself). So, if you copy commands into a text file it is a good idea to edit out the >characters at the beginning of command lines;keepingtheminwillgiveerrorsasyoucanseeinthefollowingexample:

>> apply(data, 2, mean, na.rm = TRUE) Error:
unexpected '>' in ">"

Using Annotations with the #Character

As you look through R help entries, you will see lines of explanation in the examples (not always very clear, perhaps) that are associated with the # character, also known as the hash or pound character. R essentially ignores anything that follows a # character so the best use of this character is to keep notes. For instance, in your text file you can use the # character to create annotations that help you remember what is going on. The following example shows some commands that you might use to make yourself a basic bar chart with error bars (using standard error) and used the # character to organize the information:

# Barplot with se error bars.
# make copy of data called dat. mn=apply(dat,2,mean,na.rm = TRUE) # set the meanvalues.
stdev=apply(dat,2,sd,na.rm = TRUE) # make the std deviation.
tot=apply(dat,2,sum,na.rm = TRUE) # get the sum for each column. n=mn/tot # work out the no. observations (length does not accept na.rm=T).
se=stdev/sqrt(n) # calculate std err. mx=round(max(mn+se)+0.5,0) # largest value to set y-axis. bp=barplot(mn, ylim = c(0,mx)) # make plot and set y-axis to max value.
arrows(bp,mn+se,bp,mn-se,length=0.1,angle=90,code=3) # add error bars.
If y-axis still too short change mx value to a larger one.
END

Creating Simple Functions

When you use R, you'll realize that there are a lot of commands that you can use! In spite of this, on some occasions it would be useful to have others, especially to carry out some tasks that you might require reasonably often. You can use the function() command to create new commands that you can then store and use again later. The general form of the command is like so:

function(args) expr

Insidetheparenthesesyoutypetheargumentsthatyourequireforthefunction to work; after the parentheses you type the expression you require using the argumentsyouhaveprovided.

One-Line Functions

The following example shows a simple one-line function that you can create yourself:

> log2 = function(x) log(x, base = 2)

Here you create an object called log2; the function has only one simple argument, which you call x. After the function() part you type the actual expression you want to evaluate; in this case you use the log() command using base 2. When you use your function, you type its name and give appropriate instructions inside the parentheses. In this case you require numeric input, and you see the result of the function when you type a value into the newcommand:

> log2(64) [1]
6
> log2(seq(2,8,2))
[1] 1.000000 2.000000 2.584963 3.000000
> log2(c(2,4,8,16)) [1] 1 2
3 4

The object you created as part of your new function resides in the computer memory and can be listed like other objects. You can also save your function along with the workspace when you quit R or as part of a save.image() command. Your function object is bundled and encoded along with the other objects, but this is no problem because you can retrieve the function object at any time and edit it as you require. When you use save() to save individual R objects,theyrequireafilenamewithafileextension.Dataitemsusuallyhavean
.Rdata extension and functions that you create usually get a simple .R file extension. This enables you to differentiate between the two types of objects because .Rdata items are encoded and can be opened only by R, whereas .R items are usually plain text and can be read, and perhaps edited, by other programs.

Using Default Values inFunctions

Whenyoucreateafunctionwiththefunction(),commandyougivethevarious arguments as part of the command; you can also specify default values using = and the default value. The following example uses a fairly simplemathematical equationtodeterminetheflowofwaterinastream(theManningequation);you
havethreeargumentsandoneofthemhasadefault:

manning = function(radius, gradient, coef=0.1125)
(radius^(2/3)*gradient^0.5/coef)

The three arguments are radius, gradient, and coef (the Manning coefficient). You set the coef argument to have a default value of 0.1125. If you do not specify a coef when you run the function, the following value will be used:

> manning(radius = 1, gradient = 1/500) [1] 0.3975232

You can use abbreviations when you run your new command as long as they are

unambiguous (here the arguments have completely different names):

```
> manning(gra = 1/500, ra = 1) [1]
0.3975232
```

Ofcourse,youcanoverridethe defaultvaluesforanyoftheseta rgumentslike so:

> manning(radius = 1, gradient = 1/500, coef = c(0.08, 0.11, 0.2)) [1] 0.5590170 0.4065578 0.2236068

Here you give three values for your coefargument and obtain three results.

Simple Customized Functions with Multiple Lines

A one-line script is very useful, but most of the time you will need longer and potentially more complex functions, which require several lines of commands. In that case you need a way to stop R from evaluating the function before you have finished typing it. One option would be to type the commands into a text editor and then copy and paste them into R. Another solution is to use curly brackets ({}). You use these brackets to create subsections of commands so you can use them to define the lines that form your function. The following example shows a simple function that determines the running median of a numeric vector:

```
> cummedian = function(x) {
+ tmp = seq_along(x)
+ for(i in 1:length(tmp)) tmp[i] = median(x[1:i])
+ print(tmp)
+ }
```

The first line starts the function by assigning it a name and listing the arguments; here there is only one argument, x. Rather than use any expressions at this point, you simply type a {and press the Enter key.

```
> cummedian
function(x) {
tmp = seq_along(x)
for(i in 1:length(tmp)) tmp[i] = median(x[1:i]) print(tmp)
}
```

Youcanalsousetheargs()commandtoviewtherequiredargumentsforyour function():

```
> args(manning)
function (radius, gradient, coef = 0.1125) NULL
> args(cummedian)
function  (x) NULL
```

Storing CustomizedFunctions

If you create a simple function from the command line and have created a name for it, the object will appear along with other objects when you use the ls() command. You can save your customized functions along with all the data by saving the workspace when you use quit() to quit

the program. You can also save one or several function objects using the save()command like so:

```
> save(manning, cummedian, file = 'My Functions.R')
```

In this example you save two custom functions to a file calledMy Functions.R; note that you give the file an .R extension to differentiate the file from data. The filename must be in quotes. However, when you use the save() command, R converts the object into a special binary form and you no longer have a plain textfile!

Ideally you would save your function as a plain text script so that you can edit it. You can make your function objects save to disk as plain text by using the dump() command likeso:

```
> dump(c('cummedian', 'manning'), file = 'My Functions.R')
```

If you use dump()to save your function objects, they appear as plain text and you could open and edit them with a text editor.

In most cases it is not practical to make complicated functions from the command line of R itself; it is better and easier to use a text editor. In Windows and Macintosh versions of R, editors are built-in to R and open when you make or open a script from the File menu. In Linux you must use a separate editor of your choice from the OS. If you use a text editor, you can call up the resulting plain text file from R using the source()command:

```
source(file.choose())
```

In this version of the command, you get to choose your file from a browser- like window. This option is not available in Linux OS; you must type the filename explicitly. Forexample:

```
> source(file = 'My Functions.R')
```

Making SourceCode

In addition to the usual commands that you have seen before, a few extra ones are especially useful for use with your customized functions and scripts.

Displaying the Results of Customized Functions and Scripts

When you create a custom function you may use several arguments and create new variables as part of any calculations. In the following example, which you saw earlier, you create a new variable calledtmp:

```
> cummedian
function(x) {
        tmp = seq_along(x) # a temp variable
        for(i in 1:length(tmp)) tmp[i] = median(x[1:i]) print(tmp) # the
        result
        }
```

This variable exists only while the function is being evaluated. It does not remain afterwards, as the following example shows:

```
> cummedian(mf$BOD)
 [1] 200.0 190.0 180.0 157.5 135.0 127.5 120.0 127.5 135.0 151.5
158.0 151.5 145.0
[14] 145.0 145.0 151.5 158.0 157.5 157.0 157.5 158.0 157.5 157.0
151.0 145.0
```

```
> tmp
Error: object 'tmp' not found
```

Displaying Messages as Part of ScriptOutput

You may want to produce text output as part of your script; for example, to embellish the result and make it clearer for the user. Often you will create summary statistics as part of your custom functions, and you can create text to set out the results in various forms to present them to the user more clearly. At other times you may want to pause and wait for an input from theuser.

Simple Screen Text

You can produce text on the screen to present results, or to remind the user of what was done. A simple way to do this is to use the cat() command, which enables you to present text on the screen. Your text must be in quotes, or be an object that is a character object. See the following example:

```
> msg = 'My work is far from done.'
> cat(msg)
My work is far from done.

> cat('Any text to be used must be in quotes') Any text to be used
must be in quotes
```

If you want to create new lines, you add \nto your command like so:

```
> cat('This is line 1\nThis is line 2\nThis is line 3') This is line 1
This is line 2 This
is line 3
```

Youcanhaveseveralpartstoyourcat()command,separatedbycommas.For example:

```
> cat('Am I done?\n', msg, '\n') Am I done?
  My work is far from done.
```

In the following simple script you create a data frame using some simple numericdata:

```
## Test script
                dat1 = c(1,2,4,6,7,8)
              dat2 = c(4,5,8,7,6,5)
                 dat3 = data.frame(dat1, dat2) rm(dat1,

dat2)

msg = 'My work is done.'
```

```
cat('\nOur result data is dat3:\n\n') print(dat3)
    cat('\n', msg, '\n')

## END
```

The preceding script works in the following manner:

1. Two numeric vectors are created and then joined to make a separate data frame.

2. The two individual vectors are removed to leave the data frame, called dat3, intact.

3. Asimplecharactervectoriscreatedcalledmsg.

4. Now some output is presented using the first cat() command, which consists of a single text string that starts with a new line and ends with two newlines.

5. Aprint()commandisusedtodisplaythedataframeobjectyoucreated.

6. Finally, the msg character vector is used as part of another cat() command. The text was saved to a plain text file, and to run the lines of command you use the source()command. The result is as follows:

```
> source('test script.R')
```

Our result data is: dat3

```
  dat1 dat2
1    1    4
2    2    5
3    4    8
4    6    7
5    7    6
6    8    5
```

My work is done.

>

Now take a look at the following script; in it you create a customized function; this creates cumulative results for a numeric vector. Previously you used a similar script to create a running or cumulative median; here you can specify any mathematical function, although you set the default to the median in this instance:

```
## Cumulative functions
## Mark Gardener 2011

cum.fun = function(x, fun = median, ...) { tmp =
   seq_along(x)
      for(i in 1:length(tmp)) tmp[i] = fun(x[1:i], ...)

          cat('\n', deparse(substitute(fun)),'of',
deparse(substitute(x)),'\n')
              print(tmp)
}

## END
```

In this preceding example you require two arguments, x and fun; x is the vector of numeric values and fun is the mathematical function you want to apply. You also include an ellipsis (...), which is a way of saying "allow other

instructionsthatmightberelevant.”Youmight,forexample,wanttoaddna.rm = TRUE as an instruction to take care of any NA items.

When you get the result it would be helpful to have a reminder of which function you requested when you typed the command. It would also be helpful to take the name of an object and display it as text so you can have a reminder of the data name that was used. This can be tricky though; you cannot include the data or function name here as x or fun because R will try to coerce the contents of these items as objects rather than as text (R assumes that you want to display the object itself and gives you the contents rather than the name). Additionally, you cannot put the names in quotes because they will become “fixed” and you simply get what was in the quotes. What you actually do, as you can see, is use deparse(substitute()). This looks at what you typed in the command as argumentsandconvertstheseargumentstotextobjects.

The result of the deparse(substitute()) command in the preceding script is asfollows:

```
> cum.fun(mf$BOD, mean)

  mean of mf$BOD
  [1] 200.0000 190.0000 171.6667 158.7500 149.0000 144.1667
137.1429 141.0000
  [9] 145.3333 150.3000 151.0000 150.5000 149.6923 149.3571
150.4000 152.6875
[17] 154.8824 155.0000 151.5789 155.7500 157.8571 153.1818
150.3043 148.0833
[25] 145.9600
```

Display a Message and Wait for User Intervention

There are times when you will want to pause the running of a script: this may be to give the user time to see an intermediate result (for example, a graphic) before moving on, or to provide options for the user to select: Pressing one key performs one operation and another key does somethingelse.

You can use the readline() command to accept a key press from the user; the script will wait until a key is pressed. As part of the command, you can include a message to be displayed on the screen. For example:

```
> readline(prompt = 'Press <enter> to continue:')
```

The text that follows the prompt = instruction is displayed and the script pauses until a key is pressed. Although the text in this case implies that the Enter key should be pressed, any key will do.

You can give the user options by setting an object using the readline() command. The following example could be included in a larger script:

```
yorn <- readline(prompt = "Do you want to carry on? (Y or N) :") if (yorn == 'Y' || yorn
    =='y'){
    cat('Thank goodness')
}
```

If the user presses the Y key (uppercase or lowercase), the message is displayed. If the user presses anything else, nothing happens.

You can also create user prompts that provide multiple options. Each option must have its own code within a pair of curly brackets as in the following example:

```
# Explicit options

mopts = function(){
  yorn <- readline(prompt = "Do you want to carry on? (Y or N) : ")

    if(yorn == 'Y' || yorn == 'y') { cat('Thank
        goodness')
          }


    if(yorn == 'N' || yorn =='n') { cat('Oh dear')
          }
}
## END
```

This preceding code creates a new function called mopts. When this function is run the user is presented with the text prompt. If she types one of the specified options then the appropriate message is displayed. In this case you can see that there are two options and if the user types something other than these two options then nothing will happen. You can create a catch all option by using the else command as the following exampleshows:

```
# Single positive option

sopt = function(){
  yorn <- readline(prompt = "Do you want to carry on? (Y or N) : ")

    if(yorn == 'Y' || yorn == 'y') cat('Thank goodness') else cat('Oh dear')
}
## END
```

In the following activity you create a new customized function, which you thensavetodiskforfutureuse.Thescriptcreatesabarchartofmeanvaluesand addsstandarderrorbars.Thedataneedtobeincolumnformatwithonecolumn for the numerical data (the response data) and one column for the grouping (predictor)variable.

There is, of course, a lot more to programming in R and many additional commands that you could employ. However, what you have seen here will take you a long way. By understanding more about how R works you will be able to see more and more how to customize it to carry out those tasks that are important to you.

Exercises

You can find answers to these exercises in Appendix A.

1. Make a simple function that raises a number to any power (that is, x^y). Callyourfunctionpwr(thereisalreadyacommandcalledpower).Makethe defaultraisetheinputnumber(s)tothepowerof2.

2. How can you save your new pwr function/command to disk for later recall?

3. Retype your custom pwr function/command but this time incorporate someannotations.

4. Alter your pwrfunction/command so that the user must type the required powerfromthekeyboardseparately.

5. Howcanyoumodifythepwrfunction/commandthatyoumadetopresent theuserwithasummaryofwhatwasdone?