Chapter 3

Starting Out: Working With Objects

What you will learn in this chapter:

- How to manipulate data objects
- How to select and display parts of data objects How to
- sort and rearrange data objects
- How to construct data objects
- How to determine what form a data object is
- How to convert a data object from one form to another

Data objects are the fundamental items that you work with in R. Carrying out analyses on your data and making sense of the results are the primary reasons you are using R. In this chapter you learn more about data objects and how to work with them. You learn how to recognize the different forms of data objects that R uses, and how to convert one form into another. You also learn how to sort and rearrange data, and how to extract parts of data that match criteria that you set. All of these tasks are essential in the path towards understanding R as well as being able to understand your data and analyze it effectively.

Manipulating Objects

Now that you have a few data objects to deal with it is time to think about examining these objects and getting to grips with the actual data they contain. When you collect data the first step is to get the data into R. After this you will want to look at your data, and perform summary statistics and other analyses on them. Although many analytical operations can be conducted on the data "as is," there will be many occasions when you will want to manipulate the data you have; you may want to reorder the data into a new and more informative manner, or you may want to extract certain parts of a complex data object for some further purpose. There are many ways to manipulate your data, and understanding how to do this is important in learning about R because the more you know about the way R handles objects, the better use you can make of R as an analyticaltool.

Manipulating Vectors

Vectors are essentially the building blocks of more complicated items and you can use them to construct such objects or as data items in their own right. Being able to manipulate vectors is important because they are such fundamental objects. You can manipulate vectors in several ways, all of which can be important in data analysis. The main ways you can manipulate vectors are the following:

- Selecting and displaying certain parts Sorting
- and rearranging
- Returning logical values

Later in this chapter you will also see how to combine vectors to make more complicated data objects like a matrix, a list and data frames, which all have their ownuses.

Selecting and Displaying Parts of a Vector

Being able to select and display parts of a vector can be important for many reasons. For example, if you have a large sample of data you may want to see which items are larger than a certain value. Alternatively you may want to extract a series of values as a subsample in an analysis. Being able to select parts of a vector is a basic skill that underpins many more complicated operations in R.

Here is a simple vector of numbers that form a sample:

```
> data1
 [1] 3 5 7 5 3 2 6 8 5 6 9
```

These values were the `data1` object that you created a while back. To see the entire vector you simply type its name. You can also display part of the vector using an additional part to your command as shown in Table3-1.

Table 3-1: Various Ways to Select Part of a Vector Object

Command	**Result**
`data1[1]`	Shows the first item in the vector.
`data1[3]`	Shows the third item.
`data1[1:3]`	Shows the first to the third items.
`data1[-1]`	Shows all except the first item.
`data1[c(1, 3, 4, 8)]`	Shows the items listed in the `c()` part.
`data1[data1 > 3]`	Shows all items greater than 3.
`data1[data1 < 5 | data1 > 7]`	Showsitemslessthan5orgreaterthan7.

In Table 3-1you can see how to use the square brackets after the name to select out parts of the object named.

You can use other commands on your object to help you extract various parts including thefollowing:

```
length(data1)
```

This tells you how many elements there are in the vector, including `NA`items.
You can incorporate this in the square brackets like so:

```
data1[(length(data1)-5):length(data1)]
```

This shows the last five elements of the vector.

Youcanuseotheroperations.Inthefollowingexamplethe`max()` commandis usedtogetthelargestvalueinthevector:

```
> data1
 [1] 3 5 7 5 3 2 6 8 5 6 9
>
max(data1)
[1] 9

> which(data1 ==
max(data1)) [1] 11
```

The first command, `max()`, gives the actual value that is the largest numerical value in the vector (in this case 9). The second command asks which of the elements is the largest. The value obtained is 11, meaning that the eleventh item in the vector is the largest. Notice that double equal signs are used here.

NOTE

Another useful command is one that generates sequences, `seq()`. You can use this to extract values from a vector at regular intervals. For example:

```
> data1[seq(1, length(data1), 2)]
[1] 3 7 3 6 5 9
```

This would pick out a sequence beginning with the first and ending with the last and with an interval of two. In other words, you select the first, third, fifth, and so on. You use the `length()` part to ensure that you stop once you get to the end. The general form of the `seq()` command is like so:

```
seq(start, end, interval)
```

You have to specify how far along you want to start from, where you want to end up, and how big the intervalis.

These commands will work on character vectors as well as numeric, so for example:

```
> data5
 [1] "Jan" "Feb" "Mar" "Apr" "May" "Jun" "Jul" "Aug" "Sep"
"Oct" "Nov" "Dec"
> data5[-1:-6]
[1] "Jul" "Aug" "Sep" "Oct" "Nov" "Dec"
```

Here the last six months in your vector of months are picked out. In the following example the biggest value is selected:

```
> which(data5 ==
max(data5)) [1] 9
```

In this example R has selected the ninth item, which is `"Sep"`. The items are sorted alphabetically, so the biggest is determined by that order as opposed to a numerical value.

Sorting and Rearranging a Vector

You can rearrange the items in a vector to be in one order or another using the `sort()` command. The default is to use ascending order and to leave out any `NA` items, like so:

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

>
sort(unmow)
[1] 7 8 9 9
```

In the preceding example, you can see that the numbers have been re-ordered and the `NA` item has been stripped out. You can alter the sort order by using `decreasing = TRUE`as an additional instruction in the command:

```
> sort(unmow, decreasing =
TRUE) [1] 9 9 8 7
```

You can change the way `NA` items are dealt with using the `na.last` instruction.Youhavethreeoptions:thedefaultis`NA`,meaningtheyaredropped. Ifyouuse`TRUE`the`NA`itemsareplacedlastandifyouuse`FALSE`theyareplaced first:

```
> sort(unmow, na.last =
NA) [1] 7 8 9 9
> sort(unmow, na.last =
TRUE) [1]  7   8    9     9NA

> sort(unmow, na.last =
FALSE) [1]NA  7   8 9     9
```

You can get an index using the `order()` command. This uses the same instructions as the `sort()` command, but tells you the position of each item along the vector:

```
> unmow
[1]  8   9   7   9NA
>
order(unmow)
[1] 3 1 2 4 5

> order(unmow, na.last =
NA) [1] 3 1 2 4

> order(unmow, na.last =
FALSE) [1] 5 3 1 2 4
```

The default instructions in the preceding example set `na.last =TRUE`, which is slightly different than the `sort()` command. You can also see here that the `order()` command reported that the third item in the vector is the first value when orderednumerically.

The `rank()` command sorts your data in a slightly different way than the `order()` command: it handles tied values. Compare the two commands in the followingexample:

```
> unmow
[1]  8   9   7   9
NA
>
order(unmow)
[1] 3 1 2 4 5

> rank(unmow)
[1] 2.0 3.5 1.0 3.5 5.0
```

When using the `order()` command you see that the two values of 9 in the original vector are given a different value (2 and 4). There is no reason why they could not have been ordered the other way around (that is, 4 and 2); the default is to take them as they come. When you use the `rank()` command you get a different result; the two 9 values are the third and fourth largest and by default they get a shared rank of 3.5. The `NA` item is placed at the end because by default the `na.last = TRUE`is the default setting.

You can alter the way tied values are handled using the `ties.method = "method"` instruction; by default this is set to `"average"` and this is the method used in all non-parametric statistical routines. You have other options and can set `"first"`, `"random"`, `"max"`, or `"min"` as alternatives to the default `"average"`:

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

> rank(unmow, ties.method =
'first') [1] 2 3 1 4 5

> rank(unmow, ties.method =
'average') [1] 2.0 3.5 1.0 3.5 5.0

> rank(unmow, ties.method =
'max') [1] 2 4 1 4 5

> rank(unmow, ties.method = 'random', na.last =
'keep') [1]  2   3    1    4NA
```

In the last example you can see a different option for the `na.last =` instruction; you can choose to keep any `NA`items intact. The following example also shows this:

```
> dat.na
[1]  2  5  4NA   7  3  9 NA12

> rank(dat.na, na.last =
'keep') [1]      1  4  3NA   5
     2  6NA      7
```

The `rank()` command is especially useful for non-parametric statistical testing, which relies heavily on the original data being converted to ranks.

Returning Logical Values from aVector

Previously you saw how the `which()` command was used to tell which items in a vector met some criterion. See the following code snippet for a reminder:

```
> data1
 [1] 3 5 7 5 3 2 6 8 5 6 9


> which(data1 ==
6) [1]  710
```

Here it has been determined that the seventh and tenth items are equal to 6; note that two

= signs are used like so==.

If you omit the `which()` command and use the == directly you get a different sort of answer:

```
> data1 == 6
 [1] FALSE FALSE FALSE FALSEFALSEFALSE  TRUEFALSEFALSE
                                       TRUE FALSE
```

Now you have logical answers; each item is looked at in turn to see if it is equal to 6; you get a `TRUE` or `FALSE` result for each item in the vector. You can use other mathematical operators, especially the greater than or less than symbols, like the following examples:

```
> data2 >5
 [1]FALSEFALSE   TRUE FALSEFALSEFALSE   TRUE  TRUEFALSE
                 TRUE TRUE FALSEFALSE
[14]  TRUE FALSEFALSE

> data2 <5
 [1]  TRUE FALSEFALSEFALSE   TRUE  TRUE FALSE FALSEFALSEFALSE
FALSE  TRUEFALSE
[14]FALSE   TRUE  TRUE
```

You can also combine items using various logical operators:

```
> data2 >5 & data2 <8
 [1]FALSEFALSE   TRUE FALSEFALSEFALSE   TRUEFALSEFALSE
                 TRUE FALSE FALSEFALSE
[14]  TRUE FALSEFALSE

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

> data8 == 'Feb'| data8 == 'Apr'
 [1]FALSE   TRUEFALSE   TRUE FALSE FALSE FALSE FALSEFALSEFALSE
FALSEFALSE
```

In the first example the `&`symbol was used to look for items greater than 5 AND less than 8. In the second example the pipe symbol (`|`) was used to match “`Feb`”OR “`Apr`”.

Manipulating Matrix and DataFrames

Where you have a matrix or a data frame you have a two-dimensional object rather than the one dimension of a vector. Complex objects tend to be used for many statistical operations simply because that is the nature of statistics—you rarely have anything as simple as a single column of figures.

Selecting and Displaying Parts of a Matrix or Data Frame

Like a vector, you can still use the square brackets when displaying a matrix or data frame, but now you need to specify two dimensions: `object[row, column]`. In the following activity you explore a data frame object, using the square brackets to select out various parts of a data frameobject.

Try It Out: Select Parts of a Data Frame Object

1. Look at the data frame called `mf` that contains five columns of data. To view it simply type itsname:

```
> 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
...
```

2. Pick out the item from the third row and the thirdcolumn:

```
> mf[3,3]
[1] 45
```

3. Now select the third row and display columns one tofour:

```
> mf[3,1:4]
  LengthSpeedAlgae NO3 3
      22   12
      451.75
```

Displayalltherowsbyleavingoutthefirstvalue;selectthefirst Column alone:

```
> mf[,1]
[1] 20 21 22 23 21 20 19 16 15 14 21 21 21 20 19 18 17 19 21 13 16 25 24 2322
```

1. Specify several rows but leave out a value at the end to display all columns:

```
> mf[c(1,3,5,7),]
  Length  Speed  Algae   NO3  BOD
1     20     12     40  2.25  200
3     22     12     45  1.75  135
5     21     20     75  1.95  110
7     19     17     65  1.85   95
```

2. Now specify several rows but use a `-4` to indicate that you want to displayallcolumnsexceptthefourth:

```
> mf[c(1,3,5,7),-4]
  Length  Speed  Algae  BOD
1     20     12     40  200
3     22     12     45  135
5     21     20     75  110
7     19     17     65   95
```

3. Because the columns are named you can select one by using its name rather than a simplevalue:

```
> mf[c(1,3,5,7), 'Algae'] [1]
40 45 75 65
```

4. Try giving a single value in the squarebrackets:

```
> mf[3]
   Algae
1     40
2     45
3     45
```

```
4     80
5     75
6     65
7     65
...
```

How It Works

The square brackets indicate that you want to subset the data object. The first value indicates the rows required and the second value, after a comma, indicates the columns required. If you leave out a value, all rows (or columns) are chosen. Within each part of the brackets you can use a variety of methods to select the items you require, `seq()` or `c()` commands, for example. If you use `-ve` values, then these are deleted from the display. Named columns (or rows) can be displayed by giving the names (in quotes) instead of a plainnumber.

If you specify a single value in the square brackets (rather than two), R will interpret this as a column and displays the column appropriate to the value youtyped.

Try It Out: Select Parts of a Matrix Data Object

Use the `bird` data from the `Beginning.RData` file for this activity, which you explore here by selecting out various parts using the square bracket syntax.

1. Lookatthematrixobjectcalled`bird`;simplytypeitsname:

```
> bird
               Garden  Hedgerow  Parkland  Pasture  Woodland
Blackbird          47        10        40        2         2
Chaffinch          19         3         5        0         2
GreatTit           50         0        10        7         0
HouseSparrow       46        16         8        4         0
Robin               9         3         0        0         2
SongThrush          4         0         6        0         0
```

2. You can see that there are five columns and six rows; the rows are labeled rather than having a numeric index. At first glance this appears like a data frame, so use the `str()` command to look more closely:

```
> str(bird)
 int [1:6, 1:5] 47 19 50 46 9 4 10 3 0 16 ...
 - attr(*, "dimnames")=List of 2
  ..$ : chr [1:6] "Blackbird" "Chaffinch " "Great Tit" "House Sparrow "...
  ..$ : chr [1:5] "Garden" "Hedgerow" "Parkland" "Pasture"...
```

3. Use the `class()` command to see that this is a matrix:

```
> class(bird)
[1] "matrix"
```

4. Select the second row and all thecolumns:

```
> bird[2,]
  GardenHedgerowParkland PastureWoodland19  3
                 5            0     2
```

5. Now select all rows but only the fourthcolumn:

```
> bird[,4]
     Blackbird     Chaffinch     GreatTit      HouseSparrow      Robin     Song
Thrush
              2             0             7                 4         0
0
```

6. Use named rows rather than a simple number and choose allcolumns:

```
> bird[c('Robin', 'Blackbird'),]
          Garden Hedgerow Parkland Pasture Woodland
Robin          9        3        0       0        2
Blackbird     47       10       40       2        2
```

7. Nowselectasinglerowandcolumn:

```
> bird[3,1] [1]
50
```

8. Now specify a singlevalue:

```
> bird[4] [1]
46
```

How It Works

The `str()` command shows you details of the structure of an object. The `class()` command is a useful tool because it shows you what kind of object you are dealing with.

The `[row, column]` syntax works just like it did for the data frame with one minor difference. Typing a single value into the square brackets gives a single value rather than the column that you got with the dataframe.
You can type in a named row or column as long as you make sure the name is in quotes. Notice that the items are displayed in the order you typed them, so this is also a way to rearrange an object. You can alsouse
`-ve` values to indicate rows or columns that you donot want, but note that this only works if you use numbers and not names in quotes.

When you have a matrix you use the square brackets much like you did for the data frame. In the following activity you get a chance to look at a matrix object and select parts of it using the square bracket syntax.

In a matrix the data items are indexed starting from column one and row one, then reading down each column. Look at the following example, which shows a matrix that has been constructed in "order":

```
> test.matrix
     [,1] [,2] [,3] [,4]
[1,]    1    5    9   13
[2,]    2    6   10   14
[3,]    3    7   11   15
[4,]    4    8   12   16
```

Here you can see the data in order of their "index." You read down the first column and then carry on at the top of the next and so on. This order becomes important when you want to create a brand new matrix, as you see later in the section "Making Matrix Objects").

Sorting and Rearranging a Matrix or Data Frame

```
> sort(bird)
 [1]  0  0  0  0  0  0  0  0  0  2  2  2  2  3  3  4  4  5  6  7
8  9 10 10 1619
[27] 40 46 47 50

> order(bird)
 [1]  9 12 17 20 23 24 27 28 30 19 252629    811   6 22 14 1821
16  5  71510    2
[27]13   4  1  3

> rank(bird)
 [1] 29.0 26.0 30.0 28.0 22.0 16.5 23.5 14.5  5.0 25.0 14.5  5.0
27.0 18.0 23.5
[16] 21.0  5.0 19.0  11.5  5.020.0 16.5  5.0  5.0 11.5 11.5  5.0
5.011.5   5.0
```

Remember that your matrix is essentially a single vector of data that contains information about how to split it up into rows and columns. If you want to sort, order, or rank rows or columns, you must specify them explicitly:

```
> sort(bird[,1])
  Song Thrush     Robin     Chaffinch     House Sparrow
Blackbird     Great Tit
            4         9            19                 46
47            50

> order(bird[,1])
[1] 6 5 2 4 1 3
```

In this example the first column is used as the target to sort or order. The `order()` command allows you to give additional vectors; these are used as tie- breakers to help resolve the order of items in the first vector. Compare the followingcommands:

```
> order(bird[,5])
[1] 3 4 6 1 2 5

> order(bird[,5], bird[,1])
[1] 6 4 3 5 2 1
```

If you have a data frame, you need a slightly different approach. You cannot perform a `sort()` command on an entire data frame, even if it is composed entirely of the same kind of data (that is, all numeric or all character). Take the following instance:

```
> grass2
  mow unmow
1  12     8
2  15     9
3  17     7
4  11     9
```

```
5  15    NA

> sort(grass2)
Error in `[.data.frame`(x, order(x, na.last = na.last, decreasing
= decreasing)) :
  undefined columns selected
```

You simply get an error; the command needs to operate on a single vector. You can pick out a part of your data frame to run the `sort()` command when using the standard syntax in one of the followingways:

```
> sort(grass2[1,])
  unmow mow
[1]     8  12

> sort(grass2[,1])
[1] 11 12 15 15 17

> sort(grass2[,'mow'])
[1] 11 12 15 15 17

> sort(grass2$mow)
[1] 11 12 15 15 17
```

In the first example the first row of the data frame is sorted; the subsequent examples all sort the first column but use differing syntax. The `$`, for example, is used to obtain a single vector from a list or a data frame. This syntax can also be used in the `order()` command, which works pretty much the same way as it did for the matrixobject:

```
> order(grass2)
 [1]  8  6  7  9  4  1  2  5  310

> order(grass2$mow, grass2[,2])
[1] 4 1 2 5 3

> with(grass2, order(mow,unmow))
[1] 4 1 2 5 3
```

In the first case the `order()` command takes the entire data frame, because the command did not specify any explicit columns (or rows) to order. The next two cases illustrate two ways to use the syntax to order the first column, using the second column as a tie-breaker. You can sometimes use the `$` to extract a particular part of a data item (most often a list or a data frame). In the case of the data frame you can add the name of a column after the `$` to use it as an item in its own right. In the last example you can see a new command, `with()`. This temporarily "opens up" the data frame and allows you to utilize the contents of the specified object. You look at this in a bit more detail shortly in the section "Viewing Objects within Objects").

Manipulating Lists

When you have a list the square brackets give a different result compared to other data objects you have met. The following example shows a list item; you can check its structure using the `str()` command:

```
>
str(my.list)
List of 4
 $mow   : int [1:5] 12 15 17 1115
 $ unmow: int [1:4] 8 9 7 9
 $ data3: num [1:12] 6 7 8 7 6 3 8 9 10 7 ...
 $ data7: num [1:15] 23 17 12.5 11 17 12 14.5 9 11 9 ...
```

Here there are four elements in the list; each has a name preceded by a dollar sign. The list is a one-dimensional object, so you can use only a single value in the square brackets. In other words, you can only get out one entire part of the list. Forexample:

```
> my.list[1]
$mow
[1] 12 15 17 11 15
```

You will need to use a slightly different convention to extract the elements; you must use the `$` in the name. This is the only way you can utilize the `sort()`, `order()`, or `rank()` commands on list items.

```
>
sort(my.list$mow)
[1] 11 12 15 15 17

> order(my.list$mow, my.list$unmow)
Error in order(my.list$mow, my.list$unmow) : argument
lengths differ
```

In the first case the `mow` vector is extracted from the list and a `sort()` command is performed. In the next case the `order()` command is used. This fails because the two items are of different lengths; you therefore cannot use the second item as a tie-breaker. You can, however, use the `order()` command on any of the separate items in the list:

```
>
order(my.list$unmow)
[1] 3 1 2 4
```

So, each `$` element of the list can be thought of as a vector (as you see in the next section). The elements of a list need not all be vectors, so you should examine the structure of a list object carefully (use the `str()` command) before carrying outmanipulations.

Viewing Objects withinObjects

When you create a list, matrix, or data frame you are bundling together several items. In matrix and data frame objects the items are all the same length (resulting in the rectangular object), but in a list object the items can be different lengths. Each of these objects can contain several items, but you may have noticed that these do not appear when you use the `ls()` command.

Looking Inside Complicated DataObjects

Previouslyyoulookedatsomedatacalledfw.Thedatawererepresentedasadata framewithtwocolumnslikeso:

```
> fw
  abund flow
1     9    2
2    25    3
3    15    5
4     2    9
5    14   14
6    25   24
7    24   29
8    47   34
Error: object 'abund' not found
```

The problem is that the abund and flow data are contained within the fw object and R cannot "see" them. You can use the $ to help penetrate the data and extract parts you want to view. For example, if you append $ and then the name of the required column you will view that column of data:

```
> fw$abund
[1]  92515    2 14 25 2447
```

Here the entire abund data column has been selected. It does not matter if you put spaces before or after the $, but it is probably a good habit to leave them out because it reinforces the idea that fw$abundis an object in its own right.

You can now use the square brackets as before to select out parts of the item:

```
>
fw$abund[1:4]
[1]  92515
     2
```

When the target object is a list you can use the $ to extract each part. In the following example, the mowcomponent of the list is extracted:

```
> my.list$mow
[1] 12 15 17 11 15
```

You can now subset this using the square brackets:

```
>
my.list$mow[1:4]
[1] 12 15 17 11
```

If you try this with a matrix, however, you get a quite different result:

```
> bird$Garden
Error in bird$Garden : $ operator is invalid for atomic vectors
```

This is because a matrix is essentially a single data item that has been displayed in rows and columns; recall what you saw when you tried a single value in the square brackets for a matrix. You can still extract parts of a matrix usingthenamesofthecolumns,butyouneedaslightlydifferentapproach.You canusethecolumnnameinthesquarebracketslikeso:

```
> bird[,'Garden']
     Blackbird    Chaffinch    GreatTit     HouseSparrow
                  Robin  SongThrush
            47           19           50               46        9
4
```

Notice that the column name is placed in quotes. Notice also that R cannot display the result neatly on a single row and presents the results as best as it can, given the constraints of the display.

Because there are also row names in your matrix you can use these names:

```
> bird['Robin',]
  GardenHedgerowParklandPastureWoodland 93
                0           0     2
```

You can also use the names of data frames. In the following example, the `NO3` column of the `mf`data frame is picked out:

```
> mf[,'NO3']
 [1] 2.25 2.15 1.75 1.95 1.95 2.75 1.85 1.75 1.95 2.35 2.35
 2.35
2.05 1.85 1.75
[16] 1.45 1.35 2.05 1.25 1.05 2.55 2.85 2.95 2.85
1.75
```

If your data frame has named rows rather than plain index numbers, you can use them in the same manner as you did for thematrix.

Opening Complicated DataObjects

Using the `$` to retrieve a column from a data frame is sometimes a tedious process. You can temporarily make the items within a data object available for R to “see” using the `attach()` command. This allows the columns of a data frame and the elements of a list to be viewed without the need to use the `$` sign. For example:

```
attach(my.list
) attach(mf)
```

If you try to do this for a matrix you get an error like so:

```
> attach(bird)
Error in attach(bird) :
 'attach' only works for lists, data frames and environments
```

The `attach()` command is useful because it can help reduce typing. Occasionally you may have a data object with the same name as one that you retrieve using `attach()`. If that happens you see a message similar to the following:

```
> attach(mf)
```

```
The following object(s) are masked from 'package:datasets':

    BOD
```

The mf data contains a column headed BOD. It so happens that there is a data item called BOD already in the datasets package (R provides many data examples that are used mainly in the help examples). For the time that you use attach() the older item is unavailable. When you are finished it is therefore a good idea to detach() the data. This will permit the older item to become available oncemore.

The items that are attached do not appear when you do an ls() command. You can see which items are attached using the search() command:

```
> search()
 [1]".GlobalEnv"         "package:MASS"      "mf"
 [4]"tools:RGUI"         "package:stats"     "package:graphics"
 [7]"package:grDevices""package:utils"
                         "package:datasets"
[10]"package:methods"    "Autoloads"         "package:base"
```

Here you can see the packages that are loaded, and also the mf object. Items within any of these objects are available and “seen” by R.

You can also use a command that attaches the data only transiently while you perform a single command; this is the with() command. The basic form of the command is:

```
with(object, ...)
```

Inplaceofthe . . . youcantypemoreorlessanyregularcommandsuchasthe following:

```
> Algae
Error: object 'Algae' not found

> with(mf, Algae)
 [1] 40 45 45 80 75 65 65 65 35 30 65 70 85 70 35 30 50 60 70
 25
35 85 80 80 75

> with(mf,
sum(Algae)) [1] 1460
```

In the first example you see that the Algae object is not found. In the second example the with() command is used to view the vector of data. In this case it was a bit long-winded, because you could have typed mf$Algae to achieve the same result. In the final example the sum() command is used, which simply adds up the values and gives a final total. For fairly short commands the $ is generally useful, but when you encounter long and more complicated commands, using with() can save quite a bit oftyping.

Quick Looks at Complicated DataObjects

It is useful to see what a data object consists of so that you can decide which columns or rows to utilize. You can simply type the name of an object, of course. However, this might produce a lot of output. You have already met the `str()` command; this is one way to see what an object is comprised of. You might elect to show just the first few lines of a data object; which you can do using the `head()` command. This shows the top of your data object and by default shows the first six rows. Forexample:

```
> 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
```

This can be helpful because it shows you what the columns are called and gives you an idea of what the data look like. You can also display the bottom of the data using the `tail()` command:

```
> tail(mf)
   Length Speed Algae  NO3 BOD
20     13    21    25 1.05 235
21     16    22    35 2.55 200
22     25     9    85 2.85  55
23     24    11    80 2.95  87
24     23    16    80 2.85  97
25     22    15    75 1.75  95
```

This is potentially more useful because you can also see how many rows there are in total (assuming there are no set row names). You can elect to show a different number of rows using the `n =` instruction likeso:

```
> head(bird, n = 3)
           Garden Hedgerow Parkland Pasture Woodland
Blackbird      47       10       40       2        2
Chaffinch      19        3        5       0        2
Great Tit      50        0       10       7        0
```

In this case the data are in a matrix. The command works on list objects too:

```
> head(my.list, n= 2)
$mow
[1] 12 15 17 11 15

$unmow
[1] 8 9 7 9
```

There is another way to get information about an object; you can use the `summary()` command. The output you get depends on the type of object you have. If you have a data frame, you can see some summary statistics likeso:

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

Here the `summary()` command gives some simple statistics about the numeric data called `species`. The `cut` variable is a factor and the command shows the different levels as well as the number of observations in each.

If the object you are examining is a list, you are presented with a slightly simpler summaryoutput:

```
> summary(my.list)
      LengthClass   Mode
mow     5     -none-numeric
unmow  4      -none-numeric
data312       -none-numeric
data715       -none-numeric
```

In this case you can see that there are four items of varying length. The final column shows you that they are all numbers.

The summary command also works on a matrix. In the following example the matrix contains numerical values:

```
> summary(bird.m)
     Garden          Hedgerow          Parkland
                     Pasture Woodland
 Min.    : 4.00   Min.    : 0.000   Min.    : 0.00   Min. :0.000
Min.    0
 1st Qu.:11.50   1st Qu.: 0.750   1st Qu.: 5.25   1st Qu.:0.000
1st Qu.:0

 Median :32.50   Median : 3.000   Median : 7.00   Median :1.000
Median :1
 Mean   :29.17   Mean   : 5.333   Mean   :11.50   Mean   :2.167
Mean    1
 3rd Qu.:46.75   3rd Qu.: 8.250   3rd Qu.: 9.50   3rd Qu.:3.500
3rd Qu.:2
 Max.   :50.00   Max.   :16.000   Max.   :40.00   Max.   :7.000
Max.   :2
```

Each of the named columns is summarized using basic statistics. If thematrix contains non-numerical data you get a different summary. In the following example, the matrix contains months of the year (that is, it contains character data):

```
> yr.matrix
     Qtr1  Qtr2  Qtr3  Qtr4
row1 "Jan" "Apr" "Jul""Oct"
```

```
row2 "Feb" "May" "Aug" "Nov"
row3 "Mar" "Jun" "Sep" "Dec"
```

When you use the `summary()` command you see each element listed along with how many there were in each "category"; in this case the values are all 1 because there is only one of each month:

```
> summary(yr.matrix)
  Qtr1    Qtr2    Qtr3    Qtr4
 Feb:1   Apr:1   Aug:1   Dec:1
 Jan:1   Jun:1   Jul:1   Nov:1
 Mar:1   May:1   Sep:1   Oct:1
```

If the data are a simple vector, the output you get depends on the type of item. The following examples show the results for numeric data followed by character data:

```
> summary(data4)
   Min.1stQu.    Median    Mean3rdQu.
          Max. 8.00       11.00
          12.50  13.93    17.00   23.00

> summary(data5)
   Length     Class      Mode
       12 character character
```

The`summary()`commandisdesignedtobeusedtoproducespecializedoutput as the result of other commands. So, programmers who have produced a statistical routine, for example, can create a customized layout to display the result.

It is often useful to see the column names of a complex data object, perhaps as a reminder of the variables in a complex multiple regression. R provides several ways to view and set names, as you find out in the followingsection.

Viewing and Setting Names

You may want to see just the column (or row) names contained within an object. This is often a useful means of reminding yourself of the contents of a complex data object. You may also want to alter or create row or column names. Several similar commands can do this; the following list provides examples of each:

- The most basic command that enables the viewing of column or row is `names()`. The result you get depends on the object you are looking at:

```
 > names(my.list)
 [1]"mow" "unmow"
 "data3""data7"

> names(fw)
[1] "abund" "flow"

> names(mf)
[1]"Length""Speed"    "Algae"  "NO3"     "BOD"
```

```
> names(bird)
NULL
```

The first example shows the result from looking at a list. The second and third examples are both data frames, and the last is a matrix. Notice how you do not get names for amatrix.

- You can look at row names in a similar fashion using the `row.names()` command:

```
> row.names(my.list)

> row.names(fw)
[1] "1" "2" "3" "4" "5" "6" "7" "8"

> row.names(mf)
 [1]"1" "2" "3" "4" "5" "6" "7" "8" "9" "10" "11" "12" "13"
 "14""15"
[16] "16" "17" "18" "19" "20" "21" "22" "23" "24""25"

> row.names(bird)
[1]"Blackbird" "Chaffinch" "GreatTit"
               "HouseSparrow"[5]"Robin" "Song Thrush"
```

This time the command has worked for the matrix but not for the list object. You do not even get a result with the list object.

- To amend the results from the previous example, you can modify the command and use a subtly different command, `rownames()`:

```
>
rownames(my.list)
NULL
```

You still do not get anything apparently useful, but you do at least get a result.

- You can also use `colnames()` in a similar way:

```
> colnames(mf)
[1]"Length""Speed"    "Algae"  "NO3"     "BOD"

>
colnames(my.list)
NULL

> colnames(bird)
[1]"Garden"
           "Hedgerow""Parkland""Pasture"  "Woodland"
```

You still do not get any names for the list, but the data frame and matrix names displayokay.

- There is yet another command, `dimnames()`; this looks at both the row and column names at the same time:

```
>
dimnames(my.list)
```

```
NULL

  >
  dimnames(mf
  ) [[1]]
   [1]"1" "2" "3" "4" "5" "6" "7" "8" "9" "10" "11" "12" "13"
   "14""15"
  [16] "16" "17" "18" "19" "20" "21" "22" "23" "24""25"

  [[2]]
  [1]"Length""Speed"    "Algae""NO3"     "BOD"

  >
  dimnames(bird)
  [[1]]
  [1]"Blackbird" "Chaffinch" "GreatTit"
                "HouseSparrow"[5]"Robin"
                "SongThrush"

  [[2]]
  [1]"Garden"
            "Hedgerow""Parkland""Pasture"     "Woodland"
```

The command shows the row names first and then the column names, much like the `[row, col]` instruction you used when extracting parts of objects. The command does not work on a list; rows and columns are not really appropriateforlists,whichcontainitemsofdifferinglengths.

You can use these commands to create names as well as seeing what the current names are set to, like so:

```
> names(mf)
[1]"Length""Speed"   "Algae"  "NO3"    "BOD"

names(mf) = c('len','sp', 'alg',

'no3','bod')

> names(mf)
[1]"len""sp"   "alg" "no3""bod"
```

In this example the names of the columns in the `mf` data object (a data frame) are examined/displayed. Here new names are created using the `c()` command, although any object that was a vector of characters could be used. Finally the names just set are reviewed. Remember that if you can get the names using the `names()` command, you can also set them.

You can do the same thing using the `colnames()` or `rownames()` commands:

```
> sites = c('Taw', 'Torridge', 'Ouse', 'Exe', 'Lyn',
'Brook', 'Ditch', 'Fal')
> rownames(fw) = sites
> fw
         abund flow
Taw          9    2
Torridge    25    3
Ouse        15    5
Exe          2    9
Lyn         14   14
Brook       25   24
Ditch       24   29
Fal         47   34
```

In this example a separate vector was created to contain the names (the names may be useful for other data). If you can use the command to read the names, you can use it to set the names too; so `colnames()` and `rownames()` will work on amatrix.

Youcanalsousethe`dimnames()`commandtosetbothrowandcolumnnames simultaneously;thegeneralformofthecommandislikeso:

```
dimnames(our.object) = list(rows, columns)
```

In the following example a character vector object is created for each of the rows and names; these are then used to create the names:

```
> species = c('Bbird', 'C.Finch', 'Gt.Tit', 'Sparrow',
'Robin', 'Thrush')
> habitats = c('Gdn', 'Hedge', 'Park', 'Field', 'Wood')
> dimnames(bird) = list(species, habitats)
> bird
        Gdn Hedge Park Field Wood
Bbird    47    10   40     2    2
C.Finch  19     3    5     0    2
Gt.Tit   50     0   10     7    0
Sparrow  46    16    8     4    0
Robin     9     3    0     0    2
Thrush    4     0    6     0    0
```

It would, of course, be possible to type the names in one single command by specifying them explicitly like so.

```
> dimnames(bird) = list(c('Bbird', 'C.Finch', 'Gt.Tit',
'Sparrow', 'Robin',
'Thrush'), c('Gdn', 'Hedge', 'Park', 'Field',
'Wood'))
```

YoucanresetnamesusingNULL.Yousimplyusethisinsteadofanamedobject orac`()`listoflabels:

```
>dimnames(bird) = list(NULL, habitats)
>colnames(bird) = NULL
>names(fw) = NULL
```

The`dimnames()`commandwillworkperfectlywellwithNULLforamatrix,but it will not work with a data frame. In practice it is best to keep `dimnames()`

exclusivelyformatrixobjectsandtouseanalternativeforotherobjecttypes.

The various commands associated with names are summarized in Table 3-2.

Table 3-2: Commands to View and Set Names for Data Objects

Command	**Appropriate objects**
`names()`	Works on list, matrix, and data frame
`row.names()`	Works on matrix and data frame
`rownames()`	Works on matrix and data frame
`colnames()`	Works on matrix and data frame
`dimnames(row,`	WillgetandsetnamesformatrixanddataframebutNULLonlyworksforma

Rotating Data Tables

Thusfarithasbeenassumedthatthecolumnsofdataframesandmatrixobjects are fixed. However, you can easily rotate a frame or a matrix so that the rows become the columns and the columns become the rows. To do this you use the `t()` command; think of it as short for transpose. The following examplebegins with a data frame that contains two columns of numeric data; the rows are also named:

```
> 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

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

vector:

```
> mow
[1] 12 15 17 11 15
> t(mow)
     [,1] [,2] [,3] [,4] [,5]
[1,]   12   15   17   11   15
```

Vectors are treated like columns, although when displayed they look like rows (this is a bit confusing). In any event, when you apply a `t()` command you get a matrix object as a result. You can easily convert your object to a data frame, if that is what you want, using the `as.data.frame()` command. You see in more detail how to switch an object from one kind to another in an upcoming section, “Converting from One Object Type toAnother.”

Constructing DataObjects

You have seen how to make simple vectors using the `c()` and `scan()` commands. You also used `read.table()` and `read.csv()` to import from other programs; these tend to become data frames. In the previous section you looked at ways of manipulating data objects, but now it is time turn to methods of constructing them.

You have already looked at ways to create vector objects using the `c()` and `scan()` commands (in the section “Reading and Getting Data into R”). These objects are the simplest to construct. This section introduces you to constructing other objects, including lists, data frames, and matrix objects.

Making Lists

You will usually start making a list by having several vector objects that you would like to combine in some fashion to make a complicated object (that is, something other than a simple vector). The simplest complicated object is perhaps the list and this allows you to link together several vector items of varying type or size. Lists are useful because you can tie together more or less anything to make a single object that can be used to keep project items together, for example. It is also the only way to tie together very disparateobjects.

To join several vector object together you use the `list()` command. In the following example there are five vectors; the first four are numeric and the last one is comprised ofcharacters:

```
> mow; unmow; data3; data7; data8
[1] 12 15 17 11 15
[1] 8 9 7 9
[1]  6  7  8  7  6  3  8  9 10  7  6  9
[1] 23.0 17.0 12.5 11.0 17.012.014.5    9.011.0   9.0 12.514.5
17.0  8.021.0
[1] "Jan" "Feb" "Mar" "Apr" "May" "Jun" "Jul" "Aug" "Sep" "Oct"
"Nov" "Dec"
```

Notice that you can type several commands on a single line; each one is separated using a semicolon. To make these objects into a simple list, you should do something like thefollowing:

```
> grass.list = list(mow, unmow, data3, data7, data8)
> grass.list
[[1]]
[1] 12 15 17 11 15

[[2]]
[1] 8 9 7 9

[[3]]
 [1]  6  7  8  7  6  3  8  910   7  6  9
```

```
[[4]]
 [1] 23.0 17.0 12.5 11.0 17.012.014.5    9.011.0   9.0 12.514.5
17.0  8.021.0

[[5]]
 [1] "Jan" "Feb" "Mar" "Apr" "May" "Jun" "Jul" "Aug" "Sep" "Oct"
"Nov" "Dec"
```

You simply give the names of the objects as part of a `list()` command. The list contains no names so you should use one of the commands you learned previously to make some, because you will surely forget what the data are without them:

```
> names(grass.list) = c('mow', 'unmow', 'data3', 'data7',
'months')
```

In this case the names are typed in using a `c()` command, but you might already have a character vector of names somewhere that you could use instead likeso:

```
> my.names = c('mow', 'unmow', 'data3', 'data7', 'months')
> names(grass.list) = my.names
```

Lists are the simplest of the complicated objects that you can make and are useful to allow you to keep together disparate objects as one.

Making Data Frames

A data frame is a collection of columns of data. The various columns can be different sorts so you might have several columns of numbers and several of characters. The important thing is that all the columns are the same length; in other words, the vectors that go in to make up the frame are of the same length. Any "short" vectors are padded out using NA. To make a data frame you use the `data.frame()` command:

```
my.frame = data.frame(item1, item2, item3)
```

In the parentheses you give the names of the objects that you want to use as the columns, separated with commas. If you have vectors of unequal size, you must pad out the short ones yourself. In the following example, a simple data frame is created from two samples ofnumbers:

```
> sample1 = c(5,6,9,12,8)
> sample2 = c(7,9,13,10,NA)
> sample1 ;
sample2 [1]      5
     6  912      8
[1]  7  9 13 10NA
> my.frame = data.frame(sample1, sample2)
> my.frame
  sample1 sample2
1       5       7
2       6       9
3       9      13
4      12      10
5       8      NA
```

In this case only a single `NA` item had to be typed in; notice that it does not require quotes, because it is a special object in its own right. If you had more than a few `NA` items, the typing could become quite tedious. You can use a command called `rep()` to repeat items multiple times. The general form of the commandis:

```
rep(item, times)
> response = c(5,6,9,12,8,7,9,13,10)
> predictor = c(rep('open',5), rep('closed', 4))
> response ; predictor
[1]  5  6  912   8  7  9 1310
[1]"open"   "open"   "open"   "open"   "open"   "closed""closed"
"closed"
[9] "closed"
> my.frame2 = data.frame(response, predictor)
> my.frame2
  response predictor
1         5     open
2         6     open
3         9     open
4        12     open
5         8     open
6         7   closed
7         9   closed
8        13   closed
9        10   closed
```

In this example the first five values are from one sample (which is called `open`), and the next four values are from another sample (called `closed`). The resulting data frame does not contain any `NA` values at all and the result is a neat rectangular data frame.

You can also use the `length()` command to extend or trim a vector. The following example contains two numerical vectors. If you want to make them into a data frame, they need to be the samelength.

```
> mow;unmow
[1] 12 15 17 11 15
[1] 8 9 7 9

> length(unmow) = 5
> mow;unmow
[1] 12 15 17 11 15
[1]  8  9  7  9NA

> length(unmow) = 4
> mow;unmow
[1] 12 15 17 11 15
[1] 8 9 7 9

> length(unmow) = length(mow)
> mow;unmow
[1] 12 15 17 11 15
[1]  8  9  7  9NA
```

If you set the `length()` of the vector to a value greater than its current length, `NA` items are added at the end. If you set the `length()` to a value smaller than its current length, values at the end arelost.

If you want to create a data frame from vectors of different lengths, the shorter vectors will need to be lengthened using `NA` so that all the vectors are the same length. In the preceding example the length of the longest vector was used to set the length of the shorter ones like so:

```
length(short.vector) = length(long.vector)
```

This saves time, typing, and errors. You can easily check which the longest vector is; once you know this you can use the preceding command. You can also use the up arrow to recall the last command, which you can thenedit.

Making Matrix Objects

A matrix is also a rectangular object, and the columns of a matrix are also of equal length. You can create a matrix using one of several commands, depending what objects you have to begin with.

If you have vectors of data that you want to form the columns of your matrix, you can combine them using the `cbind()` command. The following example uses the same two numeric vectors createdearlier:

```
> sample1 ;
sample2 [1]      5
     6  912      8
[1]  7  9 13 10NA
> cmat = cbind(sample1, sample2)
> cmat
     sample1 sample2
[1,]       5       7
[2,]       6       9
[3,]       9      13
[4,]      12      10
[5,]       8      NA
```

In this example you can see that the columns are named; the names are taken from the names of the vector objects that were combined. You may also want to use the samples as rows, and to do that you use the `rbind()` command in a

similar fashion:

```
> rmat = rbind(sample1, sample2)
> rmat
        [,1] [,2] [,3] [,4] [,5]
 sample     5    6    9   12    8
 sample     7    9   13   10   NA
```

Now the rows are named from the names of the vectors that were used to create the rows of the matrix. A matrix is designed to hold items all of one sort; in other words, the data must be all numbers or all characters and cannot be a mixture of both. If you try to create a matrix using a mixture, the result is that all the items are converted to characters. The following example shows a vector of simple characters, which will be used to make a new matrix along with the numeric vectors from before:

```
> sample3
[1] "a" "b" "c" "d" "e"
> mix.mat = cbind(sample1, sample2, sample3)
> mix.mat
     sample1 sample2
sample3 [1,]"5"     "7"
             "a"
[2,]"6"      "9"     "b"
[3,]"9"      "13"    "c"
[4,]"12"     "10"    "d"
[5,]"8"      NA      "e"
```

You can see that all the data items are "converted" to characters; you can also see this using the `str()` command:

```
> str(mix.mat)
 chr [1:5, 1:3] "5" "6" "9" "12" "8" "7" "9" "13" "10" NA ...
 - attr(*, "dimnames")=List of 2
  ..$ : NULL
  ..$ : chr [1:3] "sample1" "sample2" "sample3"
```

This is not an insurmountable problem but it does highlight one major difference between a matrix and a data frame. If you want to extract numbers from a "mixed" matrix, you must force the items to be numeric like so:

```
>
as.numeric(mix.mat[,1])
[1]  5  6  912   8
```

Note that the contents of the matrix have not been altered but merely extracted from the data matrix as numbers. You might attempt to overwrite a column and try to force the matrix to assume a numerical value:

```
> mix.mat[,1] = as.numeric(mix.mat[,1])
> mix.mat
     sample1 sample2 sample3
[1,] "5"     "7"     "a"
[2,] "6"     "9"     "b"
[3,] "9"     "13"    "c"
[4,] "12"    "10"    "d"
[5,] "8"     NA      "e"
```

Here is an attempt to rewrite the first column with the values "forced" as numbers. You can see that it simply does not work! The only exception is the `NA` item. This is a special R object;

the `NA` always remains as it is (note that it does not appear in quotation marks).

You have yet another method to create a matrix from scratch: the `matrix()` command. You use this for occasions where your data are in a single vector of values (either numbers or characters). Mixed items will all end up as characters. In the following example the two sample vectors you just met are made into a new object by joining them together:

```
> sample1 ;
sample2 [1]      5
     6  912      8
[1]  7  9 13 10NA
> all.samples = c(sample1, sample2)
> all.samples
  [1]  5  6  912   8  7  9 13 10NA
```

Now the `matrix()` command can be used to make a new matrix. When you do this you can choose how many rows or columns you want to make; in this case there are two samples so you would want either two rows or two columns:

```
> mat = matrix(all.samples, nrow = 2)
> mat
     [,1] [,2] [,3] [,4] [,5]
[1,]    5    9    8    9   10
[2,]    6   12    7   13   NA
```

In this example the matrix was created with two rows using the `nrow = 2` instruction. However, when you look at the original data you see that the rows created do not match with the original samples. This is because the `matrix()` command places the items in order starting at the first row and column, but fills the data column by column. You asked for rows and when you did, R worked out how many columns were required. To get the data set out in appropriate fashion, you must specify the matrix with an appropriate number of columns like so:

```
> mat = matrix(all.samples, ncol = 2)
> mat
     [,1] [,2]
[1,]    5    7
[2,]    6    9
[3,]    9   13
[4,]   12   10
[5,]    8   NA
```

Once a matrix is created you can use the `rownames()` or `colnames()` commands to create names for the rows or columns. You can also use the `dimnames()` command to make labels for both rows and columns at the same time. The `dimnames()` command can also be used as an instruction within the `matrix()` command like so:

```
> cnam = c('Sample1', 'Sample2')
> rnam = c('Site1', 'Site2', 'Site3', 'Site4', 'Site5')
> mat = matrix(all.samples, ncol = 2, dimnames = list(rnam,
cnam))
```

```
> mat
        Sample1  Sample2
Site1         5        7
Site2         6        9
Site3         9       13
Site4        12       10
Site5         8       NA
```

In this example names are created for the rows and columns as separate character vectors. This required slightly more typing but it gives greater flexibility because you might want to use the character vectors for some other purpose. You might easily have specified the names all in one command likeso:

```
> mat = matrix(all.samples, ncol = 2, dimnames =
list(c('Site1', 'Site2',
'Site3', 'Site4', 'Site5'), c('Sample1', 'Sample2')))
```

The longer command is fine but more complicated, and therefore you are more likely to make a mistake. In general it is better to use shorter commands to make a longer one, to avoiderrors.

Re-ordering Data Frames and MatrixObjects

Earlier you met the `order()` command, which was used to get an index relating to the order of items in a vector (in the section "Manipulating Objects"). You also saw this `order()` command applied to data frame and matrix objects. Additionally, you can use the `order()` command to help you take apart a matrix or data frame and rebuild it in a new order; in other words, you can re-sort the entire frame or matrix. You can also create a new frame with only some of the original columns. In either event you begin by creating an index to sort the data. In the following activity you get a chance to reorder a data frame.

Try It Out: Re-order a Data Frame and Add Additional Columns

Use the `grass2` data from the `Beginning.RData` file for this activity, which you use to reorder and add to.

1. Look at the data frame called `grass2`simply by typing its name:

```
> grass2
  mo  unm
1 12    8
2 15    9
3 17    7
4 11    9
5 15   NA
```

2. Createanindexusingthevaluesinthe`mow`column,withtiesresolved bythe`unmow`column:

```
> ii = with(grass2, order(mow,unmow))
```

3. Look at the index you justcreated:

```
> ii
[1] 4 1 2 5 3
```

4. Now create a new data frame using the sort index you justmade:

```
> grass2.resort = grass2[ii,]
> grass2.resort
  mo  unm
4 11    9
1 12    8
2 15    9
5 15   NA
3 17    7
```

5. Select a different order for the columns by specifying them in the square brackets in a neworder:

```
> grass2.resort = grass2[ii, c(2, 1)]
>
  grass2.reso
  rt unmow
  mow
4    9 11
1    8 12
2    9 15
5   NA 15
3    7 17
```

6. Nowcreateanewvectorofvalues:

```
> sheep = c(12, 14, 17, 21, 17)
> sheep
[1] 12 14 17 21 17
```

7. Finally, create a data frame that includes the original data plus the new vector you just created. Use the sort index from before:

```
> grass2.resort = with(grass2, data.frame(mow,
unmow,sheep)[ii,])
> grass2.resort
  mo  unm  she
4 11    9   21
1 12    8   12
2 15    9   14
5 15   NA   17
3 17    7   17
```

How It Works

To start off you used the `with()` command because the `mow` and `unmow` objects were inside the `grass2` data frame. You re-ordered the `mow` column and used the `unmow` column to help resolve tied values. The result of the `order()` command was saved as a result object (called `ii`) to be used in the rebuilding process. After this process note how the row names show their oldvalues.

Of course you can also omit columns or indeed add them, as long as any new vectors are of the same length as the ones in the data frame. You created a new vector of values called `sheep`. The sort index was used to create a new data frame from the two existing columns in the `grass2` data frame plus the new `sheep` vector. In this instance the `data.frame()` command is required because the `sheep` vector was not part of the original data frame beingmodified.

Matrix objects can be re-ordered in a similar fashion to data frames, but you cannot use quite the same syntax to get the columns you require. The following example shows the `bird`data you met earlier. A new sort index is created:

```
> 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

HouseSparrow       46       16        8       4        0
Robin               9        3        0       0        2
SongThrush          4        0        6       0        0
> ii =                  bird[,2], bird[,4]
> ii
[1] 6 5 2 4 1 3
```

The first column is selected as the main re-sorted column and the second and fourth columns are used as tie-breakers. A new matrix can now be created using the sort index; if you do not need to add any extra columns, you can simply specify what you want using the square brackets:

```
> bird[ii,c(5:1)]
               Woodland Pasture Parkland Hedgerow Garden
Song Thrush           0       0        6        0      4
Robin                 2       0        0        3      9
Chaffinch             2       0        5        3     19
House                 0       4        8       16     46
Blackbird             2       2       40       10     47
Great Tit             0       7       10        0     50

> Urban
[1]11   8  928   9  1
> bird.extra
               Urban
Blackbird         11
Chaffinch          8
GreatTit           9
HouseSparrow      28
Robin              9
SongThrush         1
```

The `cbind()` command can be used to make the new matrix because there are columns of data; either of the following commands will create the same result:

```
> cbind(bird, Urban)[ii,]
> cbind(bird, bird.extra)[ii,]
               Garden Hedgerow Parkland Pasture Woodland Urban
Song Thrush         4        0        6       0        0     1
Robin               9        3        0       0        2     9
```

```
Chaffinch         19      3      5      0      2      8

House Sparrow     46     16      8      4      0     28
Blackbird         47     10     40      2      2     11
Great Tit         50      0     10      7      0      9
```

In the first case the simple vector is used to form the extra column; the row names are already in place from the original `bird` matrix. In the second example the matrix object is used as the source of the additional column, and once again the row names are transferred. If the additional data were in the form of a simple data frame, the result would be the same. You can also re-order the columns as you create the new matrix simply by typing the order you want them to appear in the square brackets. For example, if you wanted to reverse their order you would type thefollowing:

```
> cbind(bird,
bird.extra)[ii,6:1]
```

Using `cbind()` is easier than the `matrix()` command because you retain the row and column names in the newly created matrix. It is possible to use the `matrix()` command, but you would then have to re-establish the names. In the following example a new matrix is created using the existing data and the new `Urban`data:

```
> matrix(c(bird, Urban), ncol = 6)[ii,]
     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]    4    0    6    0    0    1
[2,]    9    3    0    0    2    9
[3,]   19    3    5    0    2    8
[4,]   46   16    8    4    0   28
[5,]   47   10   40    2    2   11
[6,]   50    0   10    7    0    9
```

The sort index is applied to re-order the rows. You see that the names are lost; you can add them afterwards using the `rownames()` and `colnames()` commands or you might add the `dimnames =` instruction to the `matrix()` command much as you sawpreviously.

```
> ii = order(bird[3,], bird[4,], bird[1,])
> ii
[1] 5 2 4 3 1
> rbind(bird[,ii])
              Woodland Hedgerow Pasture Parkland Garden
Blackbird            2       10       2       40     47
Chaffinch            2        3       0        5     19
Great Tit            0        0       7       10     50
House Sparrow        0       16       4        8     46
Robin                2        3       0        0      9
Song Thrush          0        0       0        6      4
```

The result is that your columns have been re-ordered based on the data in the third row.

NOTE

If you want to re-order both rows and columns you must do one operation and then the other; you simply cannot do it on both at the same time.

Forms of Data Objects: Testing and Converting

R utilizes different forms of data objects. The main forms, which you have met, are vector, data frame, list, and matrix. The various forms of data objects have slightly different properties and some commands require the data to be in one particular form (the help entry for each command specifies if a certain form of object is required). You can determine the form of object you have using the `class()` command. You can convert an object from one form to another using a variety of commands that you will meet shortly. It is important to be able to determine the form of object you are dealing with and be able to convert it to another form so that you can explore the full potential of R.

Testing to See What Type of Object You Have

You can get an idea of what sort of object something is by looking at it; you met the `str()` command previously (in the section "Examining Data Structure") which does just that. You can also use the `class()` command to get a result which gives the form of object (also introduced in the section "Examining Data Structure"). It is important to know what form of object you are dealing with in order to know how to handle it. In this section you will learn how to create tests for object form using the `class()` command.

The `class()` command gives you direct information in a single category about an object form. The following command can be used as a **class test**:

```
if(any(class(test.subject) == 'test.type') == TRUE) TRUE
else FALSE
```

You simply replace the `test.subject` part with the name of the object you want to test and replace the `test.type` part with the class type you want to return as `TRUE`. Note that the `test.type`must be in quotes.

Converting from One Object Form to Another

You now know how to see what sort of object you are dealing with. You can inspect objects and work out what they are, and you can also test them with explicit commands. For day-to-day operations you can simply use the `class()` command to inspect an object, but if you are writing a script to run automatically you must get more in depth and use the programming method (the class-test). When working with programming in R, once you know what form of object you are dealing with you may want to alter it into a different form. This can be useful because some operations require your data to be in one specific form. For example, the `barplot()` command, which you will encounter later in Chapter 7, "Introduction to Graphical Analysis," requires data to be in a matrix format before the graph can be produced. Therefore, it is important to know how to convert an object into anotherform.

Convert a Matrix to a Data Frame

The matrix and data frame objects are similar in that they are both rectangular, two-dimensional objects. You can convert a matrix into a data frame using the
`as.data.frame()` command. The following example shows a numeric matrix:

```
> mat
      Sample1 Sample2
Site1       5       7
Site2       6       9
Site3       9      13
Site4      12      10
Site5       8      NA
> mat2frame = as.data.frame(mat)
> mat2frame
      Sample1 Sample2
Site1       5       7
Site2       6       9
Site3       9      13
Site4      12      10
Site5       8      NA
```

After you have converted the object you can see that it looks exactly the same as before; row and column names have been preserved. Of course you can use `str()` or `class()` commands to confirm that the object is in a new form. You can do the same thing for a character matrix. The following example shows a matrix of months of the year converted into a dataframe:

```
> yr.matrix
     Qtr1  Qtr2  Qtr3  Qtr4
row1 "Jan" "Apr"
"Jul""Oct"
row2 "Feb" "May" "Aug" "Nov"
row3 "Mar" "Jun" "Sep" "Dec"

> yr.frame = as.data.frame(yr.matrix)
> yr.frame
     Qtr1   Qtr2   Qtr3
Qtr4  row1  Jan  Apr  Jul
Oct  row2  Feb  May  Aug
Nov  row3    Mar    Jun
SepDec
```

After the conversion you see that you have converted the character items into a data frame. Each column is composed of three items that are character variables. The frame treats these as factors, which makes little practical difference. The factor data can be treated like regular character data and is easily converted using the `as.character()` command.

Convert a Data Frame into a Matrix

Similarly, you can switch a data frame into a matrix using the `as.matrix()` command. In this case, of course, if you have a data frame with mixed number and character variables the result will be a character matrix. If the data frame is all numeric, the matrix will be numeric. The following example begins with an all-numeric dataframe:

```
> grass2
```

```
  mowunmow
1  12     8
2  15     9
3  17     7
4  11     9
5  15    NA
> grass2.mat = as.matrix(grass2)
> grass2.mat
     mow unmow
[1,]  12     8
[2,]  15     9
[3,]  17     7
[4,]  11     9
[5,]  15    NA
```

The final matrix is all numeric; remember that any `NA` items are kept as they are.

The next example begins with a data frame that has two columns, one is numeric and the other is a character (in this instance actually a factor):

```
> 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
> grass.mat = as.matrix(grass)
> grass.mat
  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"
```

Here you can tell that the results are all characters because they show the quotation marks. If your data frame contains row names like the following example, you can see an important difference between the frame and the matrix. The starting point is a simple data frame. This contains two columns of numeric data and each row is named:

```
> 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
> fw.mat = as.matrix(fw)
> fw[,1]
[1]  92515    2 14 25 2447
> fw.mat[,1]
     TawTorridge      Ouse       Exe       Lyn     Brook     Ditch
Fal

47

925152142524
```

A new matrix is made from the existing data frame, and if you display each object they appear identical. However, if you display only the first column you see a difference. In the data frame you get to see just the values. If you display the matrix you see the row names as well as the data. At present this seems like a fairly trivial difference, but you need to remember that the matrix and the data frame have slightly different properties. In general, the matrix is better at dealing with named rows andcolumns.

Convert a Data Frame into a List

The following example starts with a data frame that contains several columnsofnumericdata:

```
> frame.list = as.list(mf)
> frame.list
$len
 [1] 20 21 22 23 21 20 19 16 15 14 21 21 21 20 19 18 17 19 21 13
16 25 24 23 22

$sp
 [1] 12 14 12 16 20 21 17 14 16 21 212611    9  9 11 17 15 1921
22  9 11 1615

$alg
 [1] 40 45 45 80 75 65 65 65 35 30 65 70 85 70 35 30 50 60 70 25
35 85 80 80 75

$no3
 [1] 2.25 2.15 1.75 1.95 1.95 2.75 1.85 1.75 1.95 2.35 2.35 2.35
2.05 1.85 1.75
```

```
[16] 1.45 1.35 2.05 1.25 1.05 2.55 2.85 2.95 2.85 1.75

$bod
 [1] 200 180 135 120110120    95 168 180 195 158 145 140 145165
187190157    90
[20]235200    55  87  97  95
```

The resulting list contains elements that match up to the columns of the original data frame. In this example they were all numeric, but the process also works well on data frames that contain both numeric and character columns. A data frame is by definition a rectangular object; so all the columns are the same length. This results in lists with equally sized elements; any `NA` items are transferred to thelist.

Convert a Matrix into a List

If you try to convert a matrix to a list you end up with a horrendous mess! This is because a matrix is essentially a single list of data that the matrix displays in rows and columns for your convenience. If you want to make a list from a matrix, the answer is to convert the matrix to a data frame first and then convert this data frame into a list object. In the following example the matrix is converted into a frame and then a list all in one command:

```
> yr.list = as.list(as.data.frame(yr.matrix))
```

Convert a List to Something Else

If you start with a list it is generally a bit more difficult to convert it to another type of object. The main reasons are that the list can contain items of differing length, and these can be of differing sorts (numeric, character, and so on). This is of course why the list object isuseful.

If all the items in the list happen to be the same length, it is easy to convert into a data frame using the `data.frame()` command. It does not matter if the individual items are numeric or character vectors, because the data frame can handle mixed items. The following example shows a list composed of several numericvectors:

```
> a.list
$len
 [1] 20 21 22 23 21 20 19 16 15 14 21 21 21 20 19 18 17 19 21
 13
16 25 24 23 22

$sp
 [1] 12 14 12 16 20 21 17 14 16 21 212611   9   9 11 17 15 1921
22  9 11 1615

$alg
 [1] 40 45 45 80 75 65 65 65 35 30 65 70 85 70 35 30 50 60 70
 25
35 85 80 80 75

$no3
```

```
 [1] 2.25 2.15 1.75 1.95 1.95 2.75   1.85   1.95 2.35 2.35
2.05 1.85 1.75
[16] 1.45 1.35 2.05 1.25 1.05 2.55   2.85   2.85 1.75
                                     2.95
$bod
 [1] 200 180 135 120110120   95 168 180 195 158 145 140 145165
187190157    90
[20]235200    55  87  97  95

> a.frame = data.frame(a.list)
> str(a.frame)
'data.frame': 25obs.of   5variables:
$len:int    20 21 22 23 21 20 19 16 15 14...
 $ sp:int    12 14 12 16 20 21 17 14 16 21...
 $alg: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 1.95
 2.35...
 $bod:int    200 180 135 120 110 120 95 168 180 195...
```

You can see by using the `str()` command that the new object is indeed a data frame.Ifyouwantedyourlisttobeamatrix,thesimplestoptionistoconvertto a data frame first and then make that into a matrix using the `as.matrix()` command;youcoulddothisinasinglecommandbynestinglikeso:

```
> a.matrix = as.matrix(data.frame(a.list))
```

WARNING

If you try to make a matrix object from a list in one operation you get an unusual result. It is more desirable to convert the list to a data frame first and then make the matrix from this.

If your list object contains items that are of differing lengths, it is a little trickier to get them into a different form. The simplest way is to extract the components of the list to separate vectors, pad them with `NA` items as required, and then re-assemble into a dataframe.

Youcaneasilyextracttheindividualpartsbyappendingthe`$`tothelistname. In the following example one of the vectors is extracted and assigned a new name:

```
> algae = a.list$alg
> algae
 [1] 40 45 45 80 75 65 65 65 35 30 65 70 85 70 35 30 50 60 70 25
35 85 80 80 75
```

You may try to combine the list data into a two-column data frame where one column contained the values and the other related to the group the value belonged to. In the following example there is a simple list composed of two samples. They are both numeric and are of unequal length. Use the `stack()` command to combine the values into a dataframe:

```
> grass.l
$mow
```

```
[1] 12 15 17 11 15

$unmow
[1] 8 9 7 9

> grass.stak = stack(grass.l)
> grass.stak
  values   ind
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
```

The command has taken each vector in the list and joined them together to make a column in the data frame. The second column relates to the name of the list item. Here there were two items, `mow` and `unmow`. You can see how the names of these have transferred to the data frame. The column headings are `values` and `ind`. The `values` column always contains the contents of the individual vectors from the list. The `ind` column contains the name of the vector that relates to each corresponding value. Think of this as the independent variable; this is indeed the main reason for the `stack()` command, to create a column of values (dependent variable) and a grouping column (independent variable). You can change the names to something more meaningful using the `names()` command:

```
> names(grass.stak) = c('species', 'cut')
> grass.stak
  species   cut
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
```

You can reverse the process using the `unstack()` command. When you have a simple two-column data frame, you use the command like so:

```
> unstack(grass.stak)
$mow
[1] 12 15 17 11 15

$unmow
[1] 8 9 7 9
```

When you have a more complicated situation you need to modify the command. In the following example there is a data frame with three columns; the first column holds numeric data and relates to the height of plants grown under various conditions. The second column is a character variable, a factor relating to the species grown. The final column relates to the watering treatment; in this case there are three levels of watering. Here are thedata:

```
> pw
   height    plantwater
1       9vulgaris     lo
2      11vulgaris     lo
3       6vulgaris     lo
4      14vulgaris    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
```

Youcanuseanadditionalinstructioninthe`unstack()`commandtocreatethe new two-column object. The more general form of the `unstack()` command is likeso:

```
unstack(object, form = response ~ grouping)
```

The`form=`parttellsthe`unstack()`commandwhichcolumnsofdatatoselect fromtheoriginaldataframe.Intheplantandwateringexamplethe`response`is theheight,buttherearetwochoicesregardingthe`grouping`part:

```
> unstack(pw, form = height ~ plant)
  sativa vulgaris
1      7        9
2      6       11
3      5        6
4     14       14
5     17       17
6     15       19
7     44       28
8     38       31
9     37       32

> unstack(pw, form = height ~ water)
  hi lo mid
```

```
1 28   9  14
2 31  11  17
3 32   6  19
4 44   7  14
5 38   6  17
6 37   5  15
```

```
 > pw.us = unstack(pw, form = height ~ water)
> pw.us
  hi  lo mid
1 28   9  14
2 31  11  17
3 32   6  19
4 44   7  14
5 38   6  17
6 37   5  15

> stack(pw.us, select = c(hi, lo))
   values ind
1      28  hi
2      31  hi
3      32  hi
4      44  hi
5      38  hi
6      37  hi
7       9  lo
8      11  lo
9       6  lo
10      7  lo
11      6  lo
12      5  lo
```

In this case the required columns were given in a `c()` command as part of the `select =` instruction. In the following example the columns you do *not* want are given by prefacing the name with a minus sign (–):

```
> stack(pw.us, select = -lo)
   values ind
1      28  hi
2      31  hi
3      32  hi
4      44  hi
5      38  hi
6      37  hi
7      14 mid
8      17 mid
9      19 mid
10     14 mid

11     17mid
```

```
12      15mid
```

So, list objects are tricky beasts! They are very useful though because they can "tie together" objects of different sorts.

Summary

- You can extract parts of an object using `$`or `[]`syntax.
- Objects can be reordered and sorted using `order()` and `sort()` commands. The
- `rank()` command can also sort data and is used in many statistical routines.
- Dataobjectscanbeinavarietyofforms,forexample,vector,dataframe, matrix, orlist.
- Youcanusethe`class()`commandtodeterminewhatformofitemadata objectis.
- Objects can be converted from one form to another.

Exercises

.

- **Buff tail**: 10 1 37 5 12
- **Garden bee**: 8 3 19 6 4
- **Red tail**: 18 9 1 24
- **Honeybee**: 12 13 16 9 10
- **Carder bee**: 8 27 6 32 23

Make five simple numeric vectors of these data. Now join the bee vectors together to make a data frame. Each row of the resulting frame relates to a specific plant so you could assign names to the rows.

What You Learned in This Chapter

Topic	Key Points
Constructing data objects: **`data.frame()matrix()cbind()rbind()list()`**	Complex data objects can be created using various commands. The `data.frame()` command takes 1D vectors of equal length and creates a data frame. The `matrix()` command creates a 2D matrix object from a single 1D vector. The `cbind()` and `rbind()` commands assemble a matrix, by columns or rows, from several other objects. The `list()` command creates a list from several other objects.

Summarizing data objects: **`summary()str()class()length()max()min()head()tail()`**	Objects can be summarized and viewed in a variety of ways. The `summary()` command gives a broad overview, while the `str()` command is useful to see the object structure. The type of object can be ascertained using the `class()` command. The `length()` command can be used to determine the number of items in an object. The `max()` and `min()` commands display the largest and smallest values in a numeric object. The `head()` and `tail()` commands are used to display the first or last few rows of an object.
Extracting parts and manipulating objects: **`attach() detach() with()$ [row, col]names() rownames()sort() order()rank()stack()`**	Thecontentsofcomplicateddataobjectsare not directly visible to R. To access the columns of a data frame, for example, you canusethe`attach()`command.Youcan “close” the object using `detach()`. The `with()` command enables the contents of a complicated object to be accessed temporarily. You can access elements of data objects using the `$` and `[row, col]` syntax. You can set or view the names of columns or rows using the `names()`, `rownames()` or `colnames()` commands. Objects can be rearranged using `sort()` or `order()` commands.The`rank()` command shows the relative size of numeric vectors. The`stack()`commandisusedtorecombine objects,forexampletojointwovectorsinto one and create a second (factor) vector that showstheoriginofeachobservation.
Converting objects between forms: **`as.data.frame()as.matrix()as.list()`**	Objects can be converted from one form to another using a variety of commands. For example, the `as.data.frame()` command converts an object to a data frame.

UNIT-II
Terminology

Summarizing objects: `summary()`	The`summary()`commandisageneralcommand that provides a summary of an object. If you have numerical data, then you get a numerical summary (for example, mean, max, min) but if the data are text, you get a note of how many different items you have.Using the `summary()` command on the result of many analytical routines produces a special summary suited to thekindofanalysisperformed.
Summarizing samples: `mean() median() max() min()sd() var() length() sum()quantile() fivenum()`	Numericalsamplescanbesummarizedbymany commands. Simple commands like `mean()` produce a single result (the mean), whereas others produce several. The `quantile()` command,forexample,producesfivevaluesas itsresult(thefivebasicquartiles).
Cumulative statistics: `cumsum()cummax()cummin()cumproduct()seq_along()`	Somecommandsproducecumulativevalues,for example the `cumsum()` command results in the cumulative sum of a numeric sample. The `seq_along()` command creates a simple index.These commands can be combined and usedtocreatearangeofcumulativestatistics. Forexampletherunningmean:`cumsum(my.data) / seq(along = my.data)`
Summarizing rows and columns: `colSums() colMeans()rowSums() rowMeans()apply()lapply() sapply()`	The rows and columns of two-dimensional data objectscanbesummarizedinvariousways.The `colSums()` and `rowMeans()` commands, for example, produce the sum and mean values for columns and rows, respectively.The `apply()` command is more flexible in that any function can be applied to the columns (default) or rows of a data frame or matrix. The `lapply()` and `sapply()`commandsaresimilarbutaredesigned toworkwithlistobjects.
Contingency tables and cross tabulation: `table()ftable()xtabs()`	Contingency tables can be created using the `table()` command. When the data contains several columns, a “flat” table can be produced usingthe`ftable()`command.Datacanbecross- tabulated to form contingency tables using the `xtabs()` command.Contingency tables can be reorganized into a data frame using the `as.data.frame()`command.

Table summaries: `margin.table()prop.table()addmargins()`	Tables can be summarized in exactly the same way as data frames and matrix objects by using `apply()`, for example. In addition, several commands are aimed explicitly at contingency tables. The `margin.table()` command gives sums for rows/columns. The `prop.table()` command determines the proportion that table entriesmaketowardthetotal.The`addmargins()` commandappliesanyfunctiontorows/columns Of table
Testing table objects: `is.table()` `is.matrix()class()any()is(object, “type”)inherits(object, “type”)`	You can test to see if an object is of a certain type; using `is.table()` and `is.matrix()` commands,forexample,willtestforatableand amatrix,respectively.Thesecommandsproduce a`TRUE`or`FALSE`result.The`class()`command canbeusedtovieworsetthecurrenttypeofan object. Objects can have more than one class so if a test is required the `any()` command can be used to match any of the classes that may be present.The `is()` and `inherits()` commands can extract the class of an object directly and returna`TRUE`resultiftheclassmatchesthe `“type”`.
Logic and testing: `for()if() elseany()`	The `for()` command can be used to create loops (for example, in creating cumulative statistics like a running median). The `if()` command is used to test some condition and carry out a command if the result is `TRUE`. It can add the command `else` to the end to carry out a command when the result is `FALSE`.The `any()` command can be used in testing conditions to match any item in a list.
Programming/custom functions: `any()for()if() elsefunction()`	The `any()` command enables the matching of any element in a list containing several items.The `for()` command is used to create programming loops.The `if()` command is used in logical testing of some condition and can be paired with `else` to provide an alternative.Customizedcommandscanbecreated usingthe`function()`command.
Creating sequences: `seq()`	The `seq()`command produces sequences of values.
Reading data objects: `attach()detach()with()`	Variables contained within other data objects, such as the columns of a data frame, are usually inaccessibletoR.Theycanbeaccessedusingthe `$` syntax (for example, `my.data$column`).Alternatively, the enclosing object can be “opened” using the `attach()` command. The `detach()`command closes the enclosing object.The `with()`command enables temporary access to an enclosing object.