6

Subscripting

6.1 Basics of Subscripting

For objects that contain more than one element (vectors, matrices, arrays, data frames, and lists), subscripting is used to access some or all of those elements. Besides the usual numeric subscripts, R allows the use of character or logical values for subscripting. Subscripting operations are very fast and efficient, and are often the most powerful tool for accessing and manipulating data in R. The next subsections describe the different type of subscripts supported by R, and later sections will address the issues of using subscripts for particular data types.

6.2 Numeric Subscripts

Like most computer languages, numeric subscripts can be used to access the elements of a vector, array, or list. The first element of an object has subscript 1; subscripts of 0 are silently ignored. In addition to a single number, a vector of subscripts (or, for example, a function call that returns a vector of subscripts) can be used to access multiple elements. The colon operator and the `seq` function are especially useful here; see Section 2.8.1 for details.

Negative subscripts in R extract all of the elements of an object except the ones specified in the negative subscript; thus, when using numeric subscripts, subscripts must be either all positive (or zero) or all negative (or zero).

6.3 Character Subscripts

If a subscriptable object is named, a character string or vector of character strings can be used as a subscript. Negative character subscripts are not permitted; if you need to exclude elements based on their names, the `grep`

function (Section 7.7) can be used. Like other forms of subscripting, a call to any function that returns a character string or vector of strings can be used as a subscript.

6.4 Logical Subscripts

Logical values can be used to selectively access elements of a subscriptable object, provided the size of the logical object is the same as the object (or part of the object) that is being subscripted. Elements corresponding to `TRUE` values in the logical vector will be included, and objects corresponding to `FALSE` values will not. Logical subscripting provides a very powerful and simple way to perform tasks that might otherwise require loops, while increasing the efficiency of your program as well. The first step in understanding logical subscripts is to examine the result of some logical expressions. Suppose we have a vector of numbers, and we're interested in those numbers which are more than 10. We can see where those numbers are with a simple logical expression.

```
> nums = c(12,9,8,14,7,16,3,2,9)
> nums > 10
[1]  TRUE FALSE FALSE  TRUE FALSE  TRUE FALSE FALSE FALSE
```

Like most operations in R, logical operators are vectorized; applying a logical subscript to a vector or an array will produce an object of the same size and shape as the original object. In this example, we applied a logical operation to a vector of length 10, and it returned a logical vector of length 10, with `TRUE` in each position where the value in the original vector was greater than 10, and `FALSE` elsewhere. If we use this logical vector for subscripting, it will extract the elements for which the logical vector is true:

```
> nums[nums>10]
[1] 12 14 16
```

For the closely related problem of finding the indices of these elements, R provides the `which` function, which accepts a logical vector, and returns a vector containing the subscripts of the elements for which the logical vector was true:

```
> which(nums>10)
[1] 1 4 6
```

In this simple example, the operation is the equivalent of

```
> seq(along=nums)[nums > 10]
[1] 1 4 6
```

Logical subscripts allow for modification of elements that meet a particular condition by using an appropriately subscripted object on the left-hand side

of an assignment statement. If we wanted to change the numbers in `nums` that were greater than 10 to zero, we could use

```
> nums[nums > 10] = 0
> nums
[1] 0 9 8 0 7 0 3 2 9
```

6.5 Subscripting Matrices and Arrays

Multidimensional objects like matrices introduce a new type of subscripting: the empty subscript. For a multidimensional object, subscripts can be provided for each dimension, separated by commas. For example, we would refer to the element of a matrix `x` in the fourth row and third column as `x[4,3]`. If we omit, say, the second subscript and refer to `x[4,]`, the subscripting operation will apply to the entire dimension that was omitted; in this case, all of the columns in the fourth row of `x`. Thus, accessing entire rows and columns is simple; just leave out the subscript for the dimension you're not interested in. The following examples show how this can be used:

```
> x = matrix(1:12,4,3)
> x
     [,1] [,2] [,3]
[1,]    1    5    9
[2,]    2    6   10
[3,]    3    7   11
[4,]    4    8   12
> x[,1]
[1] 1 2 3 4
> x[,c(3,1)]
     [,1] [,2]
[1,]    9    1
[2,]   10    2
[3,]   11    3
[4,]   12    4
> x[2,]
[1]  2  6 10
> x[10]
[1] 10
```

Pay careful attention to the last example, where a matrix is subscripted with a single subscript. In this case, the matrix is silently treated like a vector composed of all the columns of the matrix. While this may be useful in certain situations, you should generally use two subscripts when working with matrices.

Notice that by manipulating the order of subscripts, we can create a submatrix with rows or columns in whatever order we want. This fact coupled with the `order` function provides a method to sort a matrix or data frame in the order of any of its columns. The `order` function returns a vector of indices that will permute its input argument into sorted order. Perhaps the best way to understand `order` is to consider that `x[order(x)]` will always be identical to `sort(x)`. Suppose we wish to sort the rows of the `stack.x` matrix by increasing values of the `Air.Flow` variable. We can use `order` as follows:

```
> stack.x.a = stack.x[order(stack.x[,'Air.Flow']),]
> head(stack.x.a)
   Air.Flow Water.Temp Acid.Conc.
15       50         18         89
16       50         18         86
17       50         19         72
18       50         19         79
19       50         20         80
20       56         20         82
```

Note the comma after the call to `order`, indicating that we wish to rearrange all the columns of the matrix in the order of the specified variable. To reverse the order of the resulting sort, use the `decreasing=TRUE` argument to `order`. Although the `order` function accepts multiple arguments to allow ordering by multiple variables, it is sometimes inconvenient to have to list each such argument in the function call. For example, we might want a function which can accept a variable number of ordering variables, and which will then call `order` properly, regardless of how many arguments are used. Problems like this can be easily handled in R with the `do.call` function. The idea behind `do.call` is that it takes a list of arguments and prepares a call to a function of your choice, using the list elements as if they had been passed to the function as individual arguments. The first argument to `do.call` is a function or a character variable containing the name of a function, and the only other required argument is a list containing the arguments that should be passed to the function. Using `do.call`, we can write a function to sort the rows of a data frame by any number of its columns:

```
sortframe = function(df,...)df[do.call(order,list(...)),]
```

(When used inside a function allowing multiple unnamed arguments, the expression `list(...)` creates a list containing all the unnamed arguments.) For example, to sort the rows of the `iris` data frame by `Sepal.Length` and `Sepal.Width`, we could call `sortframe` as follows:

```
> with(iris,sortframe(iris,Sepal.Length,Sepal.Width))
    Sepal.Length Sepal.Width Petal.Length Petal.Width      Species
14           4.3         3.0          1.1         0.1       setosa
9            4.4         2.9          1.4         0.2       setosa
39           4.4         3.0          1.3         0.2       setosa
43           4.4         3.2          1.3         0.2       setosa
42           4.5         2.3          1.3         0.3       setosa
4            4.6         3.1          1.5         0.2       setosa
48           4.6         3.2          1.4         0.2       setosa
7            4.6         3.4          1.4         0.3       setosa
23           4.6         3.6          1.0         0.2       setosa
                           . . .
```

Another common operation, reversing the order of rows or columns of a matrix , can be achieved through the use of a call to the `rev` function as either the row or column subscript. For example, to create a version of the `iris` data frame whose rows are in the reverse order of the original, we could use

```
> riris = iris[rev(1:nrow(iris)),]
> head(riris)
    Sepal.Length Sepal.Width Petal.Length Petal.Width      Species
150          5.9         3.0          5.1         1.8    virginica
149          6.2         3.4          5.4         2.3    virginica
148          6.5         3.0          5.2         2.0    virginica
147          6.3         2.5          5.0         1.9    virginica
146          6.7         3.0          5.2         2.3    virginica
145          6.7         3.3          5.7         2.5    virginica
```

By default, subscripting operations reduce the dimensions of an array whenever possible. The result of this is that functions will sometimes fail when passed a single row or column from a matrix, since subscripting can potentially return a vector, even though the subscripted object is an array. To prevent this from happening the array nature of the extracted part can be retained with the `drop=FALSE` argument, which is passed along with the subscripts of the array. This example shows the effect of using this argument:

```
> x = matrix(1:12,4,3)
> x[,1]
[1] 1 2 3 4
> x[,1,drop=FALSE]
     [,1]
[1,]    1
[2,]    2
[3,]    3
[4,]    4
```

Note the "extra" comma inside the subscripting brackets – `drop=FALSE` is considered an argument to the subscripting operation. `drop=FALSE` may also prove useful if a named column loses its name when passed to a function.

Using subscripts, it's easy to selectively access any combination of rows and/or columns that you need. Suppose we want to find all of the rows in `x` for which the first column is less than 3. Since we want all the elements of these rows, we will use an empty subscript for the column (second) dimension. Once again it may be instructive to examine the subscript used for the first dimension:

```
> x[,1] < 3
[1]  TRUE  TRUE FALSE FALSE
> x[x[,1] < 3,]
     [,1] [,2] [,3]
[1,]    1    5    9
[2,]    2    6   10
```

The logical vector `x[,1] < 3` is of length 4, the number of rows in the matrix; thus, it can be used as a logical subscript for the first dimension to specify the rows we're interested in. By using the expression with an empty second subscript, we extract all of the columns for these rows.

Matrices allow an additional special form of subscripting. If a two-column matrix is used as a subscript for a matrix, the elements specified by the row and column combination of each line will be accessed. This makes it easy to create matrices from tabular values. Consider the following matrix, whose first two columns represent a row and column number, and whose last column represents a value:

```
> mat = matrix(scan(),ncol=3,byrow=TRUE)
1: 1 1 12 1 2 7 2 1 9 2 2 16 3 1 12 3 2 15
19:
Read 18 items
> mat
     [,1] [,2] [,3]
[1,]    1    1   12
[2,]    1    2    7
[3,]    2    1    9
[4,]    2    2   16
[5,]    3    1   12
[6,]    3    2   15
```

The row and column numbers found in the first two columns describe a matrix with three rows and two columns; we first create a matrix of missing values to hold the result, and then use the first two columns of the matrix as the subscript, with the third column being assigned to the new matrix:

```
> newmat = matrix(NA,3,2)
> newmat[mat[,1:2]] = mat[,3]
```

```
> newmat
     [,1] [,2]
[1,]   12    7
[2,]    9   16
[3,]   12   15
```

Any elements whose values were not specified will retain their original values, in this case a value of `NA`. See the discussion of `xtabs` in Section 8.1 for an alternative method of converting tabulated data into an R table.

6.6 Specialized Functions for Matrices

Two simple functions, while not very useful on their own, extend the power of subscripting for matrices based on the relative positions of matrix elements. The `row` function, when passed a matrix, returns a matrix of the identical dimensions with the row numbers of each element, while `col` plays the same role, but uses the column numbers. For example, consider an artificial contingency table showing the results of two different classification methods for a set of objects:

```
> method1 = c(1,1,1,1,2,2,2,2,3,3,3,3,4,4,4,4)
> method2 = c(1,2,2,3,2,2,1,3,3,3,2,4,1,4,4,3)
> tt = table(method1,method2)
> tt
       method2
method1 1 2 3 4
      1 1 2 1 0
      2 1 2 1 0
      3 0 1 2 1
      4 1 0 1 2
```

Suppose we want to extract all the off-diagonal elements. One way to think about these elements is that their row number and column numbers are different. Expressed using the `row` and `col` functions, this is equivalent to

```
> offd = row(tt) != col(tt)
> offd
      [,1]  [,2]  [,3]  [,4]
[1,] FALSE  TRUE  TRUE  TRUE
[2,]  TRUE FALSE  TRUE  TRUE
[3,]  TRUE  TRUE FALSE  TRUE
[4,]  TRUE  TRUE  TRUE FALSE
```

Since this matrix is the same size as `tt`, it can be used as a subscript to extract the off-diagonal elements:

```
> tt[offd]
 [1] 1 0 1 2 1 0 1 1 1 0 0 1
```

So, for example, we could calculate the sum of the off-diagonal elements as

```
> sum(tt[offd])
```

The R functions `lower.tri` and `upper.tri` use this technique to return a logical matrix useful in extracting the lower or upper triangular elements of a matrix. Each accepts a `diag=` argument; setting this argument to `TRUE` will set the diagonal elements of the matrix to `TRUE` along with the off-diagonal ones.

The `diag` function can be used to extract or set the diagonal elements of a matrix, or to form a matrix which has specified values on the diagonals.

6.7 Lists

Lists are the most general way to store a collection of objects in R, because there is no limitation on the mode of the objects that a list may hold. Although it hasn't been explicitly stated, one rule of subscripting in R is that subscripting will always return an object of the same mode as the object being subscripted. For matrices and vectors, this is completely natural, and should never cause confusion. But for lists, there is a subtle distinction between part of a list, and the object which that part of the list represents. As a simple example, consider a list with some names and some numbers:

```
> simple = list(a=c('fred','sam','harry'),b=c(24,17,19,22))
> mode(simple)
[1] "list"
> simple[2]
$b
[1] 24 17 19 22

> mode(simple[2])
[1] "list"
```

Although it looks as if `simple[2]` represents the vector, it's actually a list containing the vector; operations that would work on the vector will fail on this list:

```
> mean(simple[2])
[1] NA
Warning message:
argument is not numeric or logical:
       returning NA in: mean.default(simple[2])
```

R provides two convenient ways to resolve this issue. First, if the elements of the list are named, the actual contents of the elements can be accessed by separating the name of the list from the name of the element with a dollar sign (`$`). So we could get around the previous problem by referring to `simple[2]`

as `simple$b`. For interactive sessions, using the dollar sign notation is the natural way to perform operations on the elements of a list.

For those situations where the dollar sign notation would be inappropriate (for example, accessing elements through their index or through a name stored in a character variable), R provides the double bracket subscript operator. Double brackets are not restricted to respect the mode of the object they are subscripting, and will extract the actual list element from the list. So in order to find the mean of a numeric list element we could use any of these three forms:

```
> mean(simple$b)
[1] 20.5
> mean(simple[[2]])
[1] 20.5
> mean(simple[['b']])
[1] 20.5
```

The key thing to notice is that in this case, single brackets will always return a list containing the selected element(s), while double brackets will return the actual contents of selected list element. This difference can be visualized by printing the two different forms:

```
> simple[1]
$a
[1] "fred"  "sam"   "harry"

> simple[[1]]
[1] "fred"  "sam"   "harry"
```

The "`$a`" is an indication that the object being displayed is a list, with a single element named `a`, not a vector. Notice that double brackets are not appropriate for ranges of list elements; in these cases single brackets must be used. For example, to access both elements of the `simple` list, we could use `simple[c(1,2)]`, `simple[1:2]`, or `simple[c('a','b')]`, but using `simple[[1:2]]` would not produce the expected result.

6.8 Subscripting Data Frames

Since data frames are a cross between a list and a matrix, it's not surprising that both matrix and list subscripting techniques apply to data frames. One of the few differences regards the use of a single subscript; when a single subscript is used with a data frame, it behaves like a list rather than a vector, and the subscripts refer to the columns of the data frame, which are its list elements.

When using logical subscripts with data frames containing missing values, it may be necessary to remove the missing values before the logical comparison

is made, or unexpected results may occur. For example, consider this small data frame where we want to find all the rows where `b` is greater than 10:

```
> dd = data.frame(a=c(5,9,12,15,17,11),b=c(8,NA,12,10,NA,15))
> dd[dd$b > 10,]
      a  b
NA   NA NA
3    12 12
NA.1 NA NA
6    11 15
```

Along with the desired results are additional rows wherever a missing value appeared in `b`. The problem is easily remedied by using a more complex logical expression that insures missing values will generate a value of `FALSE` instead of `NA`:

```
> dd[!is.na(dd$b) & dd$b > 10,]
   a  b
3 12 12
6 11 15
```

This situation is so common that R provides the `subset` function which accepts a data frame, matrix or vector, and a logical expression as its first two arguments, and which returns a similar object containing only those elements that meet the condition of the logical expression. It insures that missing values don't get included, and, if its first argument is a data frame or matrix with named columns, it also resolves variable names inside the logical expression from the object passed as the first argument. So `subset` could be used for the previous example as follows:

```
> subset(dd,b>10)
   a  b
3 12 12
6 11 15
```

Notice that it's not necessary to use the data frame name when referring to variables in the subsetting argument. A further convenience is offered by the `select=` argument which will extract only the specified columns from the data frame passed as the first argument. The argument to `select=` is a vector of integers or variable names which correspond to the columns that are to be extracted. Unlike most other functions in R, names passed through the `select=` argument can be either quoted or unquoted. To ignore columns, their name or index number can be preceded by a negative sign (`-`). For example, consider the `LifeCycleSavings` data frame distributed with R. Suppose we want to create a data frame containing the variables `pop15` and `pop75` for those observations in the data frame for which `sr` is greater than 10. The following expression will create the data frame:

```
> some = subset(LifeCycleSavings,sr>10,select=c(pop15,pop75))
```

Since the `select=` argument works by replacing variable names with their corresponding column indices, ranges of columns can be specified using variable names:

```
> life1 = subset(LifeCycleSavings,select=pop15:dpi)
```

will extract columns starting at `pop15` and ending at `dpi`. Since these are the first three columns of the data frame, an equivalent specification would be

```
> life1 = subset(LifeCycleSavings,select=1:3)
```

Similarly, we could create a data frame like `LifeCycleSavings`, but without the `pop15` and `pop75` columns with expressions like the following:

```
> life2 = subset(LifeCycleSavings,select=c(-pop15,-pop75))
```

or

```
> life2 = subset(LifeCycleSavings,select=-c(2,3))
```

Remember that the `subset` function will always return a new data frame, matrix or vector, so it is not suited for modifying selected parts of a data frame. In those cases, the basic subscripting operations described above must be used.