1

Data in R

1.1 Modes and Classes

Every object in R contains a number of attributes to describe the nature of the information in that object. Two of the most important attributes of data in R are the mode and the class. When managing data, it is important to understand the differences among the different types of data that R supports, and when problems arise with data, the problem is often that the data is the incorrect mode or class for a particular operation.

The `mode` function returns the mode of any object in R, and the `class` function returns the class. When working with data, the most commonly encountered modes of individual objects are numeric, character, and logical. However, since data in R usually revolves around a collection of data (for example, a matrix or dataset), other modes will often be encountered. When deciding on how data should be stored in R, one important consideration has to do with the mode of the data being studied. Some objects (like matrices or other arrays) demand that all the data contained in them be of the same mode; others (like lists and data frames) allow for multiple modes within a single object.

In addition to the `mode` and `class` function, the `typeof` function can sometimes provide additional information about the type of an object, although it is not generally as useful as that returned by `mode` or `class`.

One other consideration when planning how data should be entered into R has to do with categorical data. R provides the factor class to store this type of data, and factors are automatically treated specially in statistical models and plotting functions. Values stored as factors require less storage than regular values, because R need only store each unique level once. If you examine the mode of a factor object, you'll notice that it is always numeric, even though it may display as character data, so special care is needed when working with factors. The `class` function, or one of the predicate functions described in Section 1.3 can be used to recognize factors once they are stored in R. Further information about factors can be found in Chapter 5.

Another important data type concerns dates and times. While this sort of information can be stored as a simple character representation, it is difficult to manipulate in this form. R provides several mechanisms for storing dates, including the built-in `Date`, `POSIXlt`, and `POSIXct` classes, and the contributed `chron` package. The differences among these different representations as well as information on manipulating dates and times are provided in Chapter 4.

Finally, one of the most often encountered modes of data is the list. Lists are the most flexible way of storing data in R, since they can accommodate objects of different modes and lengths. Many functions in R use lists to hold their results, and lists provide a very attractive way of accumulating information incrementally. When you've got a list and need to find the modes of the components of the list, the `sapply` function (discussed in detail in Section 8.3) can be used as shown in the following example:

```
> mylist = list(a=c(1,2,3),b=c("cat","dog","duck"),
+ d=factor("a","b","a"))
> sapply(mylist,mode)
          a           b           d
  "numeric" "character"   "numeric"
> sapply(mylist,class)
          a           b           d
  "numeric" "character"    "factor"
```

1.2 Data Storage in R

It's very rare that single values (scalars) will be the center of an R session, so one of the first questions encountered when working with data in R is what sort of object should be used to hold collections of data. The vector is the simplest way to store more than one value in R. The `c` function (mnemonic for catenate or combine) allows you to quickly enter data into R:

```
> x = c(1,2,5,10)
> x
[1]  1  2  5 10
> mode(x)
[1] "numeric"
> y = c(1,2,"cat",3)
> y
[1] "1"   "2"   "cat" "3"
> mode(y)
[1] "character"
> z = c(5,TRUE,3,7)
> z
[1] 5 1 3 7
> mode(z)
[1] "numeric"
```

Notice that when elements of different modes are combined with `c`, the mode of the resultant vector is different than that of its parts. In particular, if any of the elements are character, the other elements will be converted to characters; logical elements combined with numeric elements will be converted to numeric equivalents with `TRUE` becoming 1 and `FALSE` becoming 0. The `c` function can also be used to combine vectors:

```
> all = c(x,y,z)
> all
 [1] "1"   "2"   "5"   "10"  "1"   "2"   "cat" "3"   "5"
[10] "1"   "3"   "7"
```

Once again, since some of the elements of the combined vector have mode of character, the entire vector is converted to character.

The elements of the vector can be assigned names, which will be used when the object is displayed, and which can also be used to access elements of the vector through subscripts (Section 6.1). Names can be given when the vector is first created, or they can be added or changed after the fact using the `names` assignment function:

```
> x = c(one=1,two=2,three=3)
> x
  one   two three
    1     2     3
> x = c(1,2,3)
> x
[1] 1 2 3
> names(x) = c('one','two','three')
> x
  one   two three
    1     2     3
```

A further feature of the `names` assignment function is that it can be indexed to modify only selected elements of the names:

```
> names(x)[1:2] = c('uno','dos')
> x
  uno   dos three
    1     2     3
```

One surprising fact about vectors in R is that, in many cases if two vectors involved in an operation are not of the same length, R will recycle the values of the shorter vector in order to make the lengths compatible. This is a generalization of the fact that when a vector and a scalar are involved in an operation, R will silently repeat the scalar value to correspond to each value of the vector. So to add one to each element of a vector, a scalar value of 1 can be used:

```
> nums = 1:10
> nums + 1
 [1]  2  3  4  5  6  7  8  9 10 11
```

The same sort of thing will happen if the one operand is a vector of a different length than the other:

```
> nums = 1:10
> nums + c(1,2)
 [1]  2  4  4  6  6  8  8 10 10 12
```

Note how the values 1 and 2 are repeated in order to allow the operation to succeed. R will be silent about these kind of operations, unless the length of the longer object is not an even multiple of the length of the shorter object:

```
> nums = 1:10
> nums + c(1,2,3)
 [1]  2  4  6  5  7  9  8 10 12 11
Warning message:
longer object length
        is not a multiple of shorter object length in:
                   nums + c(1, 2, 3)
```

Notice that this is just a warning; the operation is still carried out.

Arrays are a multidimensional extension of vectors, and, like vectors, all of the objects of an array must be of the same mode. The most commonly used array in R is the matrix, a 2-dimensional array. Matrices are stored internally as vectors, with the columns of the matrix "stacked" on top of each other. The `matrix` function converts a vector to a matrix. The `nrow=` and `ncol=` arguments to `matrix` specify the number of rows and columns, respectively. If only one of these arguments is given, the other will be calculated based on the length of the input data.

Since matrices are internally stored by columns, `matrix` assumes that the input vector will be converted to a matrix by columns; the `byrow=TRUE` argument can be used to override this in the more common case where the matrix needs to be read in by rows. The mode of a matrix is simply the mode of its constituent elements; the class of a matrix will be reported as `matrix`. In addition, matrices have an attribute called `dim` which is a vector of length two containing the number of rows and columns. The `dim` function returns this vector; alternatively, individual elements can be accessed using the `nrow` or `ncol` functions.

Names can be assigned to the rows and/or columns of matrices, through the `dimnames=` argument of the `matrix` function, or after the fact through the `dimnames` or `row.names` assignment function. Since the number of rows and columns of a matrix need not be the same, the value of `dimnames` must be a list; the first element is a vector of names for the rows, and the second is a vector of names for the columns. Like vectors, these names are used for display, and can be used to access elements of the matrix through subscripting. To provide names for just one dimension of a matrix, use a value of `NULL` for the dimension for which you don't wish to provide names. For example, to create a 5 × 3 matrix of random numbers (See Section 2.2), and to name the columns `A`, `B`, and `C`, we could use statements like

```
> rmat = matrix(rnorm(15),5,3,
+               dimnames=list(NULL,c('A','B','C')))
> rmat
               A          B            C
[1,] -1.15822190 -1.1431019  0.464873841
[2,] -0.04083058  0.3705789  0.320723479
[3,] -0.25480412 -0.5972248 -0.004061773
[4,]  0.48423349 -0.8727114 -0.663439822
[5,]  1.93566841 -0.2338928 -0.605026563
```

Similarly, we could first create the matrix, then provide the dimnames separately:

```
dimnames(rmat) = list(NULL,c('A','B','C'))
```

Lists provide a way to store a variety of objects of possibly varying modes in a single R object. Note that when forming a list, the mode of each object in the list is retained:

```
> mylist = list(c(1,4,6),"dog",3,"cat",TRUE,c(9,10,11))
> mylist
[[1]]
[1] 1 4 6

[[2]]
[1] "dog"

[[3]]
[1] 3

[[4]]
[1] "cat"

[[5]]
[1] TRUE

[[6]]
[1]  9 10 11

> sapply(mylist,mode)
[1] "numeric"   "character" "numeric"   "character"
[5] "logical"   "numeric"
```

The important thing to notice about lists is that the elements of the list need not be of the same mode; the simple example provided also shows that the length of the elements need not be the same.

Like other objects in R, list elements can be named, either when the list is being created, or by using the `names` assignment function if the list already exists. The `list` function takes no keyword arguments, so list elements can be named when they are passed to the function:

```
> mylist = list(first=c(1,3,5),second=c('one','three','five'),
+               third='end')
> mylist
$first
[1] 1 3 5

$second
[1] "one"   "three" "five"

$third
[1] "end"
```

The same result can be achieved using the `names` function after creating the (unnamed) list:

```
> mylist = list(c(1,3,5),c('one','three','five'),'end')
> names(mylist) = c('first','second','third')
```

Many data analyses revolve around the idea of a dataset, a collection of related values which can be treated as a single unit. For example, you might collect information about different companies; for each company you would have a name, an industry type, the number of employees, type of health care plans offered, etc. For each of the companies you study you would have values for each of these variables. If we store the data in a matrix, with rows representing observations and columns representing variables, it would be easy to access the data, but since the modes of the variables in a dataset will often not be the same, a matrix would force, say, numeric variables to be stored as character variables. To allow the ease of indexing that a matrix would provide while accommodating different modes, R provides the data frame. A data frame is a list with the restriction that each element of the list (the variables) must be of the same length as every other element of the list. Thus, the mode of a data frame is `list`, and its class is `data.frame`. While there is some overhead for storing data in a data frame as opposed to a matrix, data frames are the preferred method for working with "observations and variables"-style datasets.

1.3 Testing for Modes and Classes

While the mode or class of an object can easily be examined through the `mode` and `class` functions, in many cases R provides a simpler way to verify whether an object has a particular mode, or is a member of a particular class. A large number of functions in R, beginning with the string "`is.`", can be used to test if an object is of a particular type. Among the many such predicate functions available in R are `is.list`, `is.factor`, `is.numeric`, `is.data.frame`, and `is.character`. These functions can be used to make sure that the data you're working with will behave the way that you expect, or that functions that you write will work properly with a variety of data.

Although R is not a true object-oriented language, many functions in R, collectively known as generic functions, will behave differently depending on the class of their arguments. For a given class, you can find out which functions will treat the class specially through the `methods` function. For more information about the object-oriented models in R, see Section 2.5.

1.4 Structure of R Objects

For simple cases such as vectors, matrices, and data frames, it's usually straightforward to determine what an object in R contains; examining the class and mode of the object, along with its length or `dim` attribute, should be sufficient to allow you to work effectively with the object. This process can conveniently be carried out for all the objects in a workspace with the `ls.str` function. However, in some cases, especially with nested lists, it can be difficult to understand how information is arranged in the object, and displaying the object in its entirety rarely elucidates the structure in these cases. The following examples are artificial, and have been kept small to reduce space, but they illustrate some strategies for getting an understanding of the structure of data in R.

Returning to an earlier example, suppose we have the following list:

```
> mylist = list(a=c(1,2,3),b=c("cat","dog","duck"),
+                d=factor("a","b","a"))
```

The `summary` function will provide the names, lengths, classes, and modes of the elements of the list:

```
> summary(mylist)
  Length Class  Mode
a 3      -none- numeric
b 3      -none- character
d 1      factor numeric
```

This provides useful information, but only looks at top-level elements of the list. If we have a list whose elements are lists, `summary` will not examine the structure of those interior lists:

```
> nestlist = list(a=list(matrix(rnorm(10),5,2),val=3),
+                 b=list(sample(letters,10),values=runif(5)),
+                 c=list(list(1:10,1:20),list(1:5,1:10)))
> summary(nestlist)
  Length Class  Mode
a 2      -none- list
b 2      -none- list
c 2      -none- list
```

In situations where direct examination provides too much detail, and `summary` or similar functions provide too little detail, the `str` function tries to provide a workable compromise. With the current example, it can be seen that `str` provides details about the nature of all the components of the object, presented in a display whose indentation provides visual cues to the structure of the object:

```
> str(nestlist)
List of 3
 $ a:List of 2
  ..$    : num [1:5, 1:2]  0.302 -1.534  1.133 -2.304  0.305
  ... ..$ val: num 3
 $ b:List of 2
  ..$       : chr [1:10] "v" "i" "e" "z" ...
  ..$ values: num [1:5] 0.438 0.696 0.722 0.164 0.435
 $ c:List of 2
  ..$ :List of 2
  .. ..$ : int [1:10] 1 2 3 4 5 6 7 8 9 10
  .. ..$ : int [1:20] 1 2 3 4 5 6 7 8 9 10 ...
  ..$ :List of 2
  .. ..$ : int [1:5] 1 2 3 4 5
  .. ..$ : int [1:10] 1 2 3 4 5 6 7 8 9 10
```

The number of elements displayed for each component is controlled by the `vec.len=` argument, and can be set to 0 to suppress any values being printed; the depth of levels displayed for each object is controlled by the `max.level=` argument, which defaults to `NA`, meaning to display whatever depth of levels is actually encountered in the object.

1.5 Conversion of Objects

To temporarily change the way an object in R behaves, a variety of conversion routines, each beginning with the string "`as.`", are provided. If it makes sense, these functions can be used to create an object equivalent to the one that you're working with, but which has a different mode or class. A simple example involves numbers which are stored as characters. This may occur when data is first entered into R, or it may arise as a side effect of some other operation.

Consider the `table` function, discussed in detail in Section 8.1. This function will return a vector of integer counts representing how many times each unique value in an object appears. The vector it returns is named, based on the unique values encountered. Suppose we use the table function on a vector of numbers, and then try to use this tabled version of the data to calculate a sum of all the values:

```
> nums = c(12,10,8,12,10,12,8,10,12,8)
> tt = table(nums)
> tt
nums
 8 10 12
 3  3  4
> names(tt)
[1] "8"  "10" "12"
> sum(names(tt) * tt)
Error in names(tt) * tt : non-numeric argument
     to binary operator
```

Since the error message suggests that `sum` was expecting a numeric vector, we can create a numeric version of `names(tt)` (without modifying the original version) using `as.numeric`:

```
> sum(as.numeric(names(tt)) * tt)
[1] 102
```

Of course, not all possible conversions make sense. If an inappropriate conversion is attempted, R will produce an error or warning message, and may generate missing values. (See Section 1.6.)

Note that the `as.` forms for many types of objects behave very differently than the function which bears the type's name. For example, notice the difference between the `list` function and the `as.list` function:

```
> x = c(1,2,3,4,5)
> list(x)
[[1]]
[1] 1 2 3 4 5

> as.list(x)
[[1]]
[1] 1

[[2]]
[1] 2

[[3]]
[1] 3
```

```
[[4]]
[1] 4

[[5]]
[1] 5
```

The `list` function creates a list (of length one) containing the argument it was passed while `as.list` converts the vector into a list of the same length as the vector.

One useful conversion that will take place automatically concerns logical variables. When a logical variable is used in a numeric context, each occurrence of `TRUE` will be treated as `1`, while values of `FALSE` will be treated as 0. Coupled with the vectorization of most functions, this allows many counting operations to be performed easily. For example, to find all the values in a vector, `x`, that are greater than 0, the expression `sum(x > 0)` could be used; the number of unequal elements in two matrices `a` and `b` could be calculated as `sum(a != b)`.

1.6 Missing Values

Missing values arise in data for a variety of reasons. The missing values may be part of the original data, or they may arise as part of a computation or conversion that takes place after you've read your data into R. In all cases, missing values are treated consistently, and will propagate across any computation that involves them, so it's important to recognize missing values as early as possible when you're working with data.

The value `NA`, without quotes, represents a missing value. You can assign a variable a value of `NA`, but to test for a missing value you must use the `is.na` function. This function will return `TRUE` if a value is missing and `FALSE` otherwise.

If a missing value occurs as the result of certain computations (for example, division by zero or taking the logarithm of a negative number), it may display as `Inf` or `NaN`. While the `is.na` function will recognize these values as missing, the `is.nan` function can be used to distinguish this type of missing value from the ordinary `NA` value.

1.7 Working with Missing Values

Many of the functions provided with R have arguments that are useful when your data contain missing values. Most of the statistical summary functions (`mean`, `var`, `sum`, `min`, `max`, etc.) accept an argument called `na.rm=`, which can be set to `TRUE` if you want missing values to be removed before the summary is calculated. For functions that don't provide such an argument, the negation operator (`!`) can be used in an expression like `x[!is.na(x)]` to create a vector which contains only the nonmissing values in `x`.

The statistical modeling functions (`lm`, `glm`, `gam`, etc.) all have an argument called `na.action=`, which allows you to specify a function that will be applied to the data frame specified by the `data=` argument before the modeling function processes the data. One very useful choice for this argument is `na.omit`, which will return a data frame with any row containing one or more missing values eliminated. Don't overlook the fact that `na.omit` can be called directly to create such a data frame independent of the modeling functions. The `complete.cases` function may also be useful to achieve the same task.

Normally, missing values are not included when a variable is made into a factor; if you want the missing values to be considered a valid factor level, use the `exclude=NULL` argument to `factor` when the factor is first created. (See Chapter 5 for more details.)

When importing data from outside sources, missing values may be represented by some string other than `NA`. In those cases, the `na.strings=` argument of `read.table` (Section 2.2) can be passed a vector of character values that should be treated as missing values. Since the `na.strings=` argument cannot be set selectively for different columns, it may sometimes be prudent to read the missing values into R in whatever form they occur, and convert them later.