S-plus: advanced

helpdesk@stat.rice.edu

August 19, 2003

Lists, Structures and DataFrames

Up to this point, we've been working with numeric manipulations in S-Plus; vectors, matrices, and mathematical operations defined with respect to these objects. However, focusing only on these things ignores S's strengths, and even renders it noncompetitive as these are things that can be done better in MATLAB. We now turn to things that are more unique to S, namely the design that recognizes that data comes in many forms, not all numeric, and that there are many standard operations that we invoke to act on data in its myriad forms. This is imbedded in what has come to be known as object-oriented programming (with the perhaps unfortunate acronym OOP). Things that you create and work with in S-Plus are objects. Each object has certain properties which are immediately apparent to S-Plus and which define in part how these objects behave and can be acted upon.

Modes of data (atomic)

The most basic properties of an object are its size and mode. As an example, consider our friend the vector after some manipulations:

```
> x <- 1:5 # the basic vector
> length(x) # how long is it?
[1] 5
> mode(x) # what mode is it? This is new.
[1] "numeric"
> y <- c(x,"All") # add a string
> y
[1] "1" "2" "3" "4" "5" "All"
> length(y)
[1] 6
> mode(y)
[1] "character"
> x <- y[1:5]
> x
[1] "1" "2" "3" "4" "5"
> mode(x)
[1] "character"
> x <- as.numeric(x)
> x
[1] 1 2 3 4 5
```

```
> y + y
Problem in y + y: Non-numeric first operand
Use traceback() to see the call stack
```

In the example above, we created a vector of numbers. S looked at the vector that we created, saw that all of the entries were numbers, and defined the mode of the vector to be `numeric`, meaning that we could perform addition, multiplication, and so on using that vector. We then took that vector of numbers and added one more element, the character string ``All'', to create a new vector. S looked at this new vector and saw that not all of the entries were numeric; one entry was of mode `character`. Now, vectors in S are assumed to have entries which are all of same mode, and this vector seems to have two, so S chooses the mode which is "less restrictive" - a number can be reasonably looked at as a character string, whereas the reverse does not apply, so S defines the mode of all entries of the vector to be `character`.
Even though the vector is not numeric, it still makes sense to ask how many entries it has, so queries as to the length of the object `y` still make sense and are answered politely. Similarly, many of the tools that we learned for forming subsets of vectors work to form new vectors or matrices or objects by referring to `y` only through the indices of the entries of `y`, and these functions work just fine. Thus, redefining `x` as the vector comprised of the first 5 elements of the vector `y` works. Note that when we do this, S looks at the mode of `y` and applies this as the mode of `x` automatically; thus, we get a vector of mode character even though in this case each of the character strings is a perfectly valid number. When something like this happens, we may want to redefine the mode of `x` to be more appropriate for the actions we wish to take. S lets us do this through various conversion routines: `as.<<modename>>(x)` will recast the mode of `x` to that specified. In this case, `as.numeric(x)` returned the vector `x` that we started with initially. This mode casting approach tries to be reasonable, but it may do something that you don't expect if you try to coerce something in a nonstandard way. For example, `as.numeric(y)` will produce a numeric vector of length 6, but the last entry, corresponding to the character string ``All'' will be rendered as a "missing value" or `NA`.

There are 5 "atomic" modes in S (we will discuss non-atomic modes shortly), corresponding to the modes that entries in a vector can have:

logical
numeric
complex
character
null.

Of these, the `null` mode is probably the least useful and the least interesting, so we shall not consider it further. As for the remaining 4, they are ordered in the hierarchy given. What this means is that if you try to form a vector of several elements of different modes, the mode of the vector will the mode of the element highest in the hierarchy. Further, some operations (such as addition) will automatically upgrade the mode status of arguments within the hierarchy – attempting to add booleans will coerce the booleans to numeric before completing the operation. To hopefully make things a bit clearer, take a look at the next few commands and see if you can predict the results line by line.

```
> x <- c(T,3)
```

```
> mode(x)
[1] "numeric"
> x <- c(T,3,4+2i)
> mode(x)
[1] "complex"
> x <- c(T,3,4+2i,"All")
> x
[1] "TRUE" "3"    "4+2i" "All"
> x <- c(T,T,F)
> as.character(x)
[1] "TRUE"  "TRUE"  "FALSE"
> as.null(x)
NULL
> x <- c(x, NULL)
> x
[1] T T F
> x <- c(1:5, NA)
> x
[1]  1  2  3  4  5 NA
> as.logical(x)
[1]  T  T  T  T  T NA
> as.logical(x) + as.logical(x)
[1]  2  2  2  2  2 NA
```

Non-atomic items: Lists and Frames

Vectors (and matrices) are objects that consist of atoms of the same mode arranged in particular ways. Lists and frames are the generalizations of these concepts – lists and frames are objects that consist of objects, possibly "nonprimitive" in nature; an atom is a primitive object that cannot be reduced further. Vectors and matrices are said to be "atomic" objects, implying amongst other things that they have uniquely defined modes and that their elements are of the same size. Lists and frames are "nonatomic" and the elements of a list or a frame can have different modes and be of different sizes. As we've seen something of how vectors work, let's take a look at a list or two to illustrate the differences:

```
> x <- c(1,2,3,4,5,NA)
> x
[1]  1  2  3  4  5 NA
> y <- c(x[1:5],"A")
> y
[1] "1" "2" "3" "4" "5" "A"
> z <- matrix(0,2,2)
> z
     [,1] [,2]
[1,]    0    0
[2,]    0    0
> mylist <- list(x,y,z)
> mylist
```

```
[[1]]:
[1]  1  2  3  4  5 NA

[[2]]:
[1] "1" "2" "3" "4" "5" "A"

[[3]]:
     [,1] [,2]
[1,]    0    0
[2,]    0    0

> length(mylist)
[1] 3
> length(mylist[[2]])
[1] 6
> dim(mylist[[3]])
[1] 2 2
> mode(mylist[[1]])
[1] "numeric"
> mode(mylist)
[1] "list"
```

Thus, we have combined in one list a numeric vector of length 6 (the first element of the list), a character vector of length 6 (the second element), and a numeric matrix of dimension 2 by 2 (the third element). The list is an object, so it has a length, 3, and a mode, `list`. Note the new mode! Elements of a list can be referred to in a manner similar to that of referring to the elements of a vector. The distinction is that you use a double set of square brackets, `[[]]`, to reference the elements of a list as opposed to a single set for elements of a vector. Lists are perhaps most useful as ways of passing or returning several arguments of different modes or sizes to or from a function.

Names

An additional feature of objects is that becomes very useful when working with vectors, matrices, lists or frames, is that objects and occasionally elements of objects can have names. For example, we might want to return one vector of residuals, one vector of the raw data, and one vector of the coefficients employed, and have them named appropriately. To assign names to the elements of a list, we use the function `names` with a vector of character strings as follows:

```
> names(mylist) <- c("resids","rawdata","mycoefs")
> mylist
$resids:
[1]  1  2  3  4  5 NA

$rawdata:
[1] "1" "2" "3" "4" "5" "A"

$mycoefs:
```

```
     [,1] [,2]
[1,]    0    0
[2,]    0    0

> mylist$rawdata
[1] "1" "2" "3" "4" "5" "A"
> mylist[[2]]
[1] "1" "2" "3" "4" "5" "A"
> names(mylist) <- c("resids","raw data","mycoefs")
> mylist
$resids:
[1]  1  2  3  4  5 NA

$"raw data":
[1] "1" "2" "3" "4" "5" "A"

$mycoefs:
     [,1] [,2]
[1,]    0    0
[2,]    0    0
```

Once elements in a list have been named, you can refer to them by name using the dollar sign, as in `mylist$resids`. Names can have spaces in them, but for this to work the quotation marks indicating the beginning and the end of the string must be retained, as in `mylist$''raw data''`. Working with names can be used to make the conceptual thrust of your code and answers clearer. It is also possible to name the elements of a list at the time the list is created:

```
mylist <- list(resids=x, rawdata=y, mycoefs=z)
```

produces the same result as the first use of names above. You can also name the components of a vector, buy you cannot refer to them by name.

```
> x
[1]  1  2  3  4  5 NA
> names(x) <- c("A","B","C","D","E","F")
> x
 A B C D E  F
 1 2 3 4 5 NA
> x$A
NULL
```

Matrices have so many elements that naming all of them was deemed counterproductive, but it is possible to name the rows and columns of a matrix, giving names to each of the dimensions. This uses the `dimnames` command and a list comprised of two vectors of character strings, one for the row names and one for the column names (in that order). As with vectors, you may not refer to the elements of a matrix just using the names.

```
> z
     [,1] [,2]
```

```
[1,]    0    0
[2,]    0    0
> dimnames(z) = list(c("row1","row2"),c("col1","col2"))
> z
     col1 col2
row1    0    0
row2    0    0
> z[row1,col1]
Problem: Object "row1" not found
Use traceback() to see the call stack
```

The dimension names can also be supplied in the initial call to the matrix, as in

```
> matrix(0,2,2,dimnames=list(c("row1","row2"),c("col1","col2")))
```

We note briefly that `dimnames` assumes that you are working with an object having more than one dimension, such as a matrix, a data frame (see below) or a multidimensional array. Trying to use `dimnames` to name the elements of a vector or a list will not work. Similarly, the functions `dim`, `ncol` and `nrow` only give sensible answers when their arguments are matrices, frames, or multidimensional arrays; calling any of these functions with a singly-dimensioned argument produces `NULL`.

If you want to change the names of the rows or columns of a matrix without looking at everything, there are a few options available. First, there is the function `row.names`. Interestingly, there is no corresponding function `col.names`. Second, you can reference the appropriate entry of the first or second list element of `dimnames`.

```
> row.names(z) <- c("r1","r2")
> z
   col1 col2
r1    0    0
r2    0    0
> col.names(z) <- c("c1","c2")
Problem: couldn't find assignment function for "col.names"
> dimnames(z)
[[1]]:
[1] "r1" "r2"

[[2]]:
[1] "col1" "col2"

> dimnames(z)[[2]]
[1] "col1" "col2"
> dimnames(z)[[2]] <- c("c1","c2")
> dimnames(z)
[[1]]:
[1] "r1" "r2"

[[2]]:
[1] "c1" "c2"
```

```
> dimnames(z)[[1]][2]
[1] "r2"
```

Data Frames

Data frames extend the idea of lists, allowing for multiple modes, reference by name, and so on, while imposing the structure of a matrix so that the data can be arranged in a rectangular manner with a fixed number of rows and columns. This is reasonable in the statistical data setting, where we can think of each row denoting an object on which observations are being made, and each column as an individual covariate (x_i or y_i) where the covariates need not all be numeric. In many cases the covariates are all numeric, in which case the frame can be coerced into a matrix and vice versa.

There are some notable ways (in addition to the allowance for columns of differing modes) in which data frames differ from matrices. Let's look at a few.

```
> 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
> zed <- data.frame(x)
> zed
  x.1 x.2 x.3
1   1   5   9
2   2   6  10
3   3   7  11
4   4   8  12
> zed$x.1
[1] 1 2 3 4
> zed$1
Problem: Syntax error: illegal literal ("1") on input line 1
> zed[[1]]
[1] 1 2 3 4
> x[1:5]
[1] 1 2 3 4 5
> zed[1:5]
Problem in "[.data.frame"(x, ..1): undefined columns selected
Use traceback() to see the call stack
> zed[c(2,3)]
  x.2 x.3
1   5   9
2   6  10
3   7  11
4   8  12
> zed[1,]
```

```
  x.1 x.2 x.3
1   1   5   9
> mode(zed)
[1] "list"
```

The first difference is that data frames, unlike matrices, *insist* that their rows and columns have names. If they do not have names in a matrix which is coerced into a data frame, then S will assign default names, using numbers for the row names, and the name of the initial matrix followed by ".column number" for the column names. If the initial matrix had dimnames, these will be retained. Further, if a data frame is coerced into matrix form, it carries the dimnames with it. The second difference is that, like lists, parts of the frames can be referred to by name. This only applies to the column names! Trying to use list notation with row names will not work. This is because a data frame is a special type of list, where the columns are the elements of the list. This interpretation also underlies some of the differences that arises when you try to access the elements of a matrix or a data frame in a nonstandard way. If you invoke `x[1:5]` above, S coerces the matrix `x` into a vector and produces the first 5 elements. If you try the same thing with `zed`, which is a data frame, S notes that a data frame is also a list and tries to give you the first 5 elements of the list, which would correspond to the first 5 columns of `zed`. In this case, as there are less than 5 columns in the matrix, this produces an error. Calling `zed[c(2,3)]`, on the other hand, produces the second and third elements of the list, which do exist. Attempting to reference the elements of `zed` using the full matrix form, eg `zed[1,3]` will work (it produces 9 here).

Overwriting existing functions

S assumes that you know what you're doing when you define functions. Generally, this is a good thing, but there's one way in which this is carried too far: If you define a function and give it the same name as a function that already exists in S, S accepts the new function as the one to invoke. The latest version of S-Plus will bother to tell you that it already has a function by that name; if you see this warning, move your function! If you later try to invoke the S function, it will find yours instead, which can lead to much frustration in trying to debug code that looks correct. Let's see what happens when I overwrite the matrix transpose function.

```
> x <- matrix(1:6,2,3)
> x
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
> t(x)
     [,1] [,2]
[1,]    1    2
[2,]    3    4
[3,]    5    6
> t <- function(x){
+ x
+ }
Warning messages:
```

```
  Conflicting definitions of "t" on databases ".Data" and  "splus"
> t(x)
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
> rm(t)
> t(x)
     [,1] [,2]
[1,]    1    2
[2,]    3    4
[3,]    5    6
```

Getting data from files part II: scan and read.table

Earlier, we talked about reading lines of text into S-Plus using the `source` command, which interprets the lines of the text file supplied as valid S commands and attempts to execute them, one by one. While this is useful, and is probably the best way to work on functions, it is not the most efficient method for loading tables of data. For these purposes, the functions `scan` and `read.table` are better. The distinction is that `scan` is designed to read in the contents of a matrix, and hence looks for things to be of all the same mode, whereas `read.table` is designed to read in a data frame. In what follows, I will work with the contents of the file `dummy.q`.

Scan reads entries from a file in a continuous stream, happily ignoring whitespace save for the purpose of indicating the separation between elements of the file.

```
> !more dummy.q
1 2 3 4 5
6 7 8
> x <- matrix(scan("dummy.q"),2,4)
> x
     [,1] [,2] [,3] [,4]
[1,]    1    3    5    7
[2,]    2    4    6    8
> x <- matrix(scan("dummy.q"),2,4,byrow=T)
> x
     [,1] [,2] [,3] [,4]
[1,]    1    2    3    4
[2,]    5    6    7    8
```

In this case, we loaded the file into a matrix (it was very useful to know the number of rows and columns beforehand). We note once again that S reads matrices in by column, which is almost never the desired behavior when we are scanning data from a file.

Occasionally, a file comes with headings for the appropriate columns. Unfortunately, `scan` does not provide for automatically loading the column names, but it can be adjusted to skip over them:

```
> !more dummy.q
col1 col2 col3 col4
```

```
1 2 3 4
5 6 7 8
> x <- matrix(scan("dummy.q",skip=1),2,4,byrow=T)
> x
     [,1] [,2] [,3] [,4]
[1,]    1    2    3    4
[2,]    5    6    7    8
```

The optional argument `skip`, supplied to `scan` (not to `matrix`!) will tell scan to skip the specified number of lines in the file. There are various other options available for use with the `scan` function, but at this point I will suggest that you invoke `?scan` if you wish to know more (check into how to use symbols other than whitespace to denote item separation).

The `read.table` function is quite similar to scan, but it allows (surprise, surprise) for data that are of different modes. Unlike `scan`, `read.table` treats each row of the file as a distinct unit, so rows of the data frame that is to result may not be split across rows in the file.

```
> !more dummy.q
1 2 3 4
5 6 7 8
> zed <- read.table("dummy.q")
> zed
  V1 V2 V3 V4
1  1  2  3  4
2  5  6  7  8
> mode(zed)
[1] "list"
> zed$V1
[1] 1 5
```

Note that the result of `read.table` is of mode `list`, and that default row and column labels have been supplied. We can also deal with a "header row" in a slightly more sane way than is allowed by `scan`:

```
> zed <- read.table("dummy.q",header=T)
> zed
  col1 col2 col3 col4
1    1    2    3    4
2    5    6    7    8
```

Now, in some data files the rows are also labelled (for example, in medical files each row may denote a patient and each column some type of measurement). This can be accomodated (assuming the row label column also has an entry in the header line).

```
> !more dummy.q
rowlabels col1 col2 col3 col4
row1 1 2 3 4
row2 5 6 7 8
> zed <- read.table("dummy.q",header=T,row.names="rowlabels")
> zed
     col1 col2 col3 col4
```

```
row1    1    2    3    4
row2    5    6    7    8
```

In general, S will determine the mode of each entry column by column automatically, so you don't have to worry about it.

Statistical Models and lm

Okay, we've just spent a goodly amount of time discussing the types of data structures that are employed in S, leading up to the idea of a data frame. The reason that a data frame is important is that data frames are often supplied as local databases to the functions in S which work with statistical models. There are several such functions, but for the moment we will just be focusing on one: the `lm`, or "linear model" function. As you might guess, this function is quite pertinent to our course.

We begin by examining the relationship between `fuel` and `weight` that was the subject of the last homework. Initially, both `fuel` and `weight` exist in my directory as vectors of numbers.

```
> zed <- lm(fuel ~ weight)
> zed
Call:
lm(formula = fuel ~ weight)

Coefficients:
 (Intercept)     weight
   0.3914324 0.00131638

Degrees of freedom: 60 total; 58 residual
Residual standard error: 0.3877015
> mode(zed)
[1] "list"
> class(zed)
[1] "lm"
> names(zed)
 [1] "coefficients"  "residuals"     "fitted.values" "effects"
 [5] "R"             "rank"          "assign"        "df.residual"
 [9] "contrasts"     "terms"         "call"
```

As you might suspect, the call to `lm` fits the linear model

$$fuel = \beta_0 + \beta_1 weight + error$$

with the errors assumed to be independent and identically distributed normal random variates. Note that what the function `lm` returns is a "linear model object", as opposed to a standard list. This distinction shows up in the fact that while the `mode` of `zed` is a list, so that it can have many entries of differing modes, the `class` of `zed` is `lm`. Structures in S having different classes can have methods written for them that behave if different ways. One such function is the one that tells S to display all of an object if that object's name is typed at the prompt. This function does work precisely that way for objects of the

"numeric" or "list" class, for example, but the "lm" class has a different version invoked. This is an instance of function overloading, another hallmark of the OOP paradigm. For more on constructing classes, see the section on OOP in chapter 4 of Venables and Ripley. In any event, when S fits a linear model, it produces lots of results and doesn't show you all of them, just the ones it deems likely to be of most immediate interest. While you can "look under the hood" by invoking the appropriate elements of the `zed` list, it pays to be careful here as some of these elements are not what you might expect. The `coefficients` are what you would expect, the `residuals` are the straight residuals with no standardization, the `fitted values` are again what you would expect but most of the others are not. Rather than trying to look directly at the components, there are a variety of helper functions that have been designed to operate on the elements of the `lm` object so as to return the information of interest to you in what is most likely to be a user-readable form. In particular,

```
> print(zed)
Call:
lm(formula = fuel ~ weight)

Coefficients:
 (Intercept)     weight
   0.3914324 0.00131638

Degrees of freedom: 60 total; 58 residual
Residual standard error: 0.3877015
> summary(zed)

Call: lm(formula = fuel ~ weight)
Residuals:
     Min      1Q  Median     3Q    Max
 -0.7957 -0.2703 0.01414 0.2547 0.9583

Coefficients:
              Value Std. Error t value Pr(>|t|)
(Intercept)  0.3914  0.2995     1.3070  0.1964
     weight  0.0013  0.0001    12.9323  0.0000

Residual standard error: 0.3877 on 58 degrees of freedom
Multiple R-Squared: 0.7425
F-statistic: 167.2 on 1 and 58 degrees of freedom, the p-value is 0

Correlation of Coefficients:
       (Intercept)
weight -0.9859
> anova(zed)
Analysis of Variance Table

Response: fuel
```

```
Terms added sequentially (first to last)
          Df Sum of Sq  Mean Sq  F Value Pr(F)
   weight  1  25.13875 25.13875 167.2433     0
Residuals 58   8.71812  0.15031
> coef(zed)
 (Intercept)     weight
   0.3914324 0.00131638
> resid(zed)
```

where I omitted the printing of the vector of residuals (all 60 of them). In any event, many of the basic linear model results are available using the `lm` function. How to get at these results is described in Chapter 6 of Venables and Ripley, and we'll be spending a good deal of time talking about what the various tools are and how to use them.

How do the statistical model functions involve data frames? Well, the simple answer is that in passing arguments to `lm`, the first argument is the `formula` describing what model is to be fitted, and the optional second argument `data` will accept a data frame, causing S to look at the columns of the data frame as the covariates to be acted upon. This clusters all of the functionality of interest in a single location, which is a form of "encapsulation" (which a short way of arguing against leaving things lying around). Thus, an equivalent way of conducting the above analysis is

```
> cardata <- data.frame(cbind(fuel,weight))
> names(cardata)
[1] "fuel"   "weight"
> zed2 <- lm(fuel ~ weight, data = cardata)
> zed2
Call:
lm(formula = fuel ~ weight, data = cardata)

Coefficients:
 (Intercept)     weight
   0.3914324 0.00131638

Degrees of freedom: 60 total; 58 residual
Residual standard error: 0.3877015
```

Aside from having too many things lying around, are there other reasons for working with data frames? That depends on how difficult the data are to access initially. In this example, the gain is not significant. However, if there are lots of variables lying around and some of them are actually subsets of larger sets of interest, it can be useful to focus in on things by putting everything you want to work with into a single frame. If that sounds like a wishy-washy answer, that's because it is. The primary reason is aesthetic.

Working with Formulas: The Wilkinson-Rogers Notation

tilde indicates y on the left, x's on the right

the period by itself means include every main effect

Addition means inclusion of terms, not summation

Subtraction indicates specific exclusion

Colon indicates interaction terms, not sequences

Multiplication indicates inclusion of all higher order terms with repetitions of a given term excluded

Exponentiation indicates inclusion of higher order terms up to a specified order

The I notation says “evaluate me first” and treat formula operators as you would under normal addition rules.

you can apply exponentiation, addition, and subtraction to the dot.