S-Plus: An Introduction

helpdesk@stat.rice.edu

August 19, 2003

What is S-Plus?

The S programming language was first introduced by Richard Becker and John Chambers of AT&T, as "an Interactive Environment for Data Analysis and Graphics" in 1984. Shortly thereafter, more functionality was added, additional programming was developed (primarily by Allan Wilks) and the ensemble was re-released as "New S". Eventually, the ownership of S migrated to a company called MathSoft, which added more functionality and quite a few customized specialty packages and re-released it as S-Plus. Many years and quite a bit of academic advertising later, S-Plus has become probably the premier software package for statistical data analysis *in academia*. In industry, the dominant package is probably SAS. Neither dominance is particularly deserved, but we acknowledge the status quo by focusing on S-Plus.

In writing this, I am trying to give you a quick introduction to the basics of the S-Plus Programming Language in enough detail for you to make S do some of your bidding. This includes introducing the syntax of S as a language, discussing how to create and manipulate vectors and matrices, but not how to manipulate data frames in complete generality (that will come in a later lesson). All of the discussion below focuses on the operation of S in a UNIX environment such as Owlnet. The PC version, `S-Plus 2000`, is available at no charge for current students and can be obtained from the IT department. The PC version streamlines many of the object-handling procedures and provides familiar Windows point-and-click interface.

An additional resource on `S-plus` is *Modern Applied Statistics with S-Plus* (3rd ed, 1999) by Bill Venables and Brian Ripley. This book was used as a suplementary textbook for a couple of courses last year, so many of the second year students should have a copy.

There is a moderated S-News mailing list, which is archived at

`http://www.biostat.wustl.edu/s-news/`

1 Starting S-Plus: Invocation and .Data

Starting S-Plus is easy; at the command prompt simply type

```
> Splus
```

and you will get a response like

```
S-PLUS : Copyright (c) 1988, 2000 MathSoft, Inc.
S : Copyright Lucent Technologies, Inc.
Version 6.0 Release 1 for Sun SPARC, SunOS 5.6 : 2000
Working data will be in /home/vfofanov/MySwork
>
```

The key thing to notice here is the last statement, about where the working data will be stored. When S-Plus is invoked it looks in the current directory for a subdirectory named ".Data". The prefix dot means that a cursory UNIX `ls` command issued in that directory will not show the .Data directory; it is a hidden file. Invoking `ls -a` will show the dot files. If the current directory does not have a .Data subdirectory, S-Plus will look for a .Data subdirectory in your home directory, and if this is present it will default to using this directory as the location in which working data will be stored.

If no .Data subdirectory exists in your home directory, prevous versions of S-Plus would create a subdirectory in your home directory, `Schapter****`, where the asterisk indicates a numeric wildcard, and would create a .Data subdirectory in this new subdirectory and store the working output there. Every time S-Plus is invoked and it cannot find a .Data subdirectory in the current directory or in the home directory it will create a new `Schapter****` subdirectory.

The version of S-plus we are using (`version 6.0`) will not create `Schapter*****` by default; instead it always creates `MySwork` folder in your home directory. This eliminates multiple `Schapter****` directories but runs the risk of having data and functions from multiple unrelated sessions being stored in the same place with little clue as to their origins. You can bypass this feature and create multiple .Data folders by using the `CHAPTER` flag when invoking `S-plus`.

On a side note, the rest of the tutorial can be completed using `S-plus` invoked at a command line. However, you can now invoke `S-plus` from emacs, which you learned in the previous tutorial. This will make your job easier by colorcoding keywords and commands, and keeping track of ")". You can quit your current instance of `S-plus` by typing `q()` or `exit()`.

2 Basic units of data: vectors and scalars

2.1 Creating vectors and scalars

Splus can be used like a very expensive calculator; it will happily perform various arithmetic operations on numeric values. E.g.,

```
> sqrt(5)
[1] 2.236068
> 12*4;
[1] 48
> 12*4
[1] 48
> 12*4; sqrt(3)
[1] 48
```

```
[1] 1.732051
```

Note that including a semicolon at the end of a statement has no effect on its operation; the numeric value is still returned. The semicolon is useful for two things: for putting multiple statements on a single line (something that I'm not that fond of) and as a visual guide to humans as to where the end of a statement is (as opposed to `C`, where the semicolon is required). This latter effect is something I encourage when writing new code.

The "`[1]`" at the start of each Splus line indicates that this number is the first element of the answer. This is relevant because unlike `C` and like `MATLAB`, the basic unit of data in `S-Plus` is a vector and as such, a scalar is treated as a vector of length one. Of course, since we have very cheap calculators, we would like Splus to do more. In particular we would like to assign labels to mathematical structures that we create and to operate on these structures by invoking the labels alone. For example, we may want to create a vector of numbers called "x". This is done through the use of the Splus assignment operators, the left arrow, "`<-`", the underscore, "_" and the equals sign, "`=`". For example,

```
> x <- c(1,2,3)
> x
[1] 1 2 3
> x <- 1:40
> x
[1] 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
[26] 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40
> x _ c(1,2,3)
> x
[1] 1 2 3
> x = 1:3
> x
[1] 1 2 3
```

There are several things that should be noted about the above code.

- The first line invokes the "`c`" or "concatenate" operator which is Splus' notation that a vector of data is about to be supplied and assigns the vector $(1,2,3)$ to `x`. Note that in defining the vector, separate elements must be separated by commas.
- Unlike arithmetic operations, assignment does not produce a further printout. Typing the name of an object causes S to display the components assigned to that object.
- The colon operator is a shortcut for generating a sequence of numbers, altering by 1 at each step, from the first limit until the second limit is reached. This means that one can increment from fractional values (`1.5:4.3` produces `1.5 2.5 3.5`). If the second limit is less than the first, the sequence will decrement (`3.4:2` produces `3.4 2.4`).
- The numbering of elements within a vector starts from `1`, not from `0` as in some other languages (e.g. C). This means that if you want to access the first element of the vector `x`, you would type "`> x[1]`".

2.2 Asignment operator

While the left-pointing arrow notation for assignment is fairly self-explanatory, the underscore notation is definitely not. It is an invitation to write obscure code, and should be avoided. The reason that I mention it at all is to prevent the situation where some poor student tries `my_results <- 5`, creating two objects, `my` and `results`, both of which are equal to 5. Note that if there are multiple assignments in a single line (a bad idea), evaluation proceeds from right to left, so `my <- results <- 6` first assigns 6 to `results` and then assigns the value of `results` to `my`.

Earlier versions of S did not use the equals sign for assignment, and the proprietors still contend that the use of the left arrow notation is more descriptive. Unfortunately for them, I disagree. The use of the equals sign for assignment is sufficiently pervasive (in pretty much every other programming language of which I am aware) that I think allowing it is a welcome step. The main reason I will not use it throughout is that, as noted, it will not work with old versions of S-Plus.

2.3 Manipulating vectors

Having defined a vector, operations can now be perfomed on the vector or on some subset of it. These operations can be arithmetic, boolean, or referencing. For example, let's manipulate the vector `x` a bit.

```
> x <- 1:10
> 2*x
[1] 2 4 6 8 10 12 14 16 18 20
> x/3
[1] 0.3333333 0.6666667 1.0000000 1.3333333 1.6666667 2.0000000 2.3333333
[8] 2.6666667 3.0000000 3.3333333
> x + x
[1] 2 4 6 8 10 12 14 16 18 20
> x[1:5] + x[6:10]
[1] 7 9 11 13 15
> x[1:4] + x[3:5]
Problem in x[1:4] + x[3:5]:  length of longer operand (4) should be a multiple
of length of shorter (3)
Use traceback() to see the call stack
> x[1:4] + x[5:6]
[1] 6 8 8 10
> x[1,3:5,7:9,2:5]
Problem in [:  No dim attribute for array subset:  c(1, 2, 3, 4, 5, 6, 7, 8,
9, 10)
Use traceback() to see the call stack
> x[c(1,3:5,7:9,2:5)]
[1] 1 3 4 5 7 8 9 2 3 4 5
> x > 4
[1] F F F F T T T T T T
> x != 7
[1] T T T T T T F T T T
```

```
> x[x != 7]
[1] 1 2 3 4 5 6 8 9 10
> x[c(T,T)]
[1] 1 2 3 4 5 6 7 8 9 10
> x[c(T,F)]
[1] 1 3 5 7 9
> x[c(T,F,T)]
[1] 1 3 4 6 7 9 10
> x[c(T,T,T,T,T,T,T,T,T,T,T)]
[1] 1 2 3 4 5 6 7 8 9 10 NA
```

Applying an arithmetic operation to `x` (addition, multiplication, square root) corresponds to applying that operation to all elements of `x` in turn. Adding one vector to another (as in `x + x`) adds elementwise; first element to first element, second to second, and so on. This is the source of S's claim to being a vectorized language. Note that S does not differentiate between row and column vectors (a bit of intellectual laziness, in my opinion). Elements of `x` can be referred to using the square bracket notation. Note that if several elements of `x` are requested, S expects a vector of the desired elements within the square brackets, so using `c()` to concatenate the entries in the square brackets may be required. The vector of indices placed in the square backets may be of greater length than the vector `x` itself; the new vector created will be as long as the vector of indices.

An interesting phenomenon occurs if you try to add or multiply two vectors of different lengths. This will return an error unless the length of one of the vectors is an integer multiple of the length of the other. In this case, the shorter vector is replicated until it reaches the length of the longer and then the operation is performed. Thus, `x[1:4] + x[5:6]` corresponds to (1 2 3 4) + (5 6 5 6). This "vector filling" can be useful but also tricky to spot if it gives rise to an effect that you did not expect.

Boolean operations (`<, <=, >, >=, ==, !=`) applied to a vector return a vector of booleans. A vector of booleans can be used as an index list to elicit elements of a vector; every "true" element is returned. If used in this fashion, the length of the boolean vector used as the index list is important. If the boolean vector is shorter than `x`, it will be replicated until it is the length of `x` and those elements indicated "T" will be returned. Note that this expansion occurs even if the length of `x` is not an integer multiple of the length of the boolean vector. If the boolean vector is longer than the length of `x`, then "NA" will be returned for the last few values. similarly, attempting to reference `x[11]` if the length of `x` is 10 will yield NA. Indexing vectors are rounded towards zero, so fractional values can be used (`x[2.6]` returns `x[2]`). Trying to reference `x[0]` returns `numeric(0)`. Referencing negative values does something completely different; `x[-k]` returns the vector x without the values given in `k`:

```
> k <- c(1,2,11,2,5)
> x[-k]
[1] 3 4 6 7 8 9 10
```

Values specified twice in `k` are only omitted once, and if `k` contains values in excess of the length of `x`, nothing is omitted in correspondence.

Query: how might you return every third element of a vector?

Boolean vectors may be combined using and (`&&` or `&`), or (`||` or `|`), and not (`!`) in their various combinations. The different symbols do have different meanings! The doubled forms of `and` and `or` are used with expressions returning a single boolean value, not a vector of booleans. These are the versions most often used in the definitions of loop conditions. The single versions act on vectors of booleans in a componentwise fashion:

```
> x = 1:3
> ((x > 1) && (!(x == 3)))
[1] F
Warning messages:
Condition has 3 elements:  only the first used in:  e1 && e2
> ((x > 1) & (!(x == 3)))
[1] F T F
```

Boolean vectors can be coerced into behaving like integers, with True yielding 1 and False yielding 0; conversely, numbers can be compared to booleans with 0 denoting False and nonzero numbers denoting True. For example,

```
> 5 + (x > 1)
[1] 5 6 6
> (c(0,0,0) | (!(x == 3)))
[1] T T F
> (c(1,2,3) | (!(x == 3)))
[1] T T T
> (0 | (!(x == 3)))
[1] T T F
> (0 & (!(x == 3)))
[1] F F F
```

As the last two lines illustrate, the comparators also use vector extension, but they again require the length of the longer vector to be an integer multiple of the length of the shorter. The not operator is fairly straightforward, but occasionally needs to be explicitly bound by parentheses – an exclamation point at the start of a command line in S-Plus indicates that the following line is to be fed as a command to the underlying UNIX shell. Thus,

```
> !(x == 3)
x:  Command not found
> (!(x == 3))
[1] T T F
```

3 Functions

3.1 Defining functions: syntax

Another feature that separates a programming language from a glorified calculator is that we can write new functions to operate on objects. We'll begin by writing a simple function, one that computes the dot product of two vectors.

```
> x = 1:3
> y = 4:6
> dot <- function(x,y) sum(x * y)
> dot(x,y)
[1] 32
```

This illustrates some of the basic syntax – we begin with the name of the function, and we assign a "function" to that name with a list of arguments to be specified. Following that, we provide the statements that operate on those arguments to yield the result. In this case, there was just a single statement, so we could write it on the same line and be done with it. As this will not generally be the case, we would like to be able to extend things over multiple lines. This is done using syntactically incomplete statements. For example, we could have defined the dot product above using

```
> dot = function(x,y){
+ sum(x*y);
+ }
> dot(x,y)
[1] 32
> dot
function(x, y)
{
    sum(x * y)
}
>
```

The open brackets indicate that we are in the process of defining a function, and when we hit enter before typing the closed bracket, the statement is syntactically incomplete. S waits for completion before operating, and it indicates that it is waiting for completion by changing the prompt from ">" to "+". If you have a statement (anywhere, not just in the definition of a function) that is too long for one line, it can be extended onto the next solely by leaving it syntactically incomplete. Finally, we note that once a function is defined, typing the name of the function without providing any arguments provides a listing of the body of the function, with statement separating semicolons and any comments stripped away, and with the statements indented to show the order of nesting. This automatic indentation is a good thing; I approve of it.

One very nice thing about S as a programming language is that it provides for an easy way of supplying default values for function arguments.

```
> dot = function(x,y = x){
+ sum(x*y);
+ }
> dot(x,y)
[1] 32
> dot(x,x)
[1] 14
> dot(x)
[1] 14
```

In the redefined version of dot, if the argument `y` is left unspecified in the call to dot, it assumes the value of `x` as a default. Thus, `dot(x)` is the same as `dot(x,x)`. This corresponds roughly to the idea of function overloading, or allowing variable numbers of arguments. Note, though, that the default only kicks in if an a second argument is not supplied. If a second argument is supplied it overrides the default.

It is also useful to note that since the type of the object is not specified in the declaration of the function it is possible to run into problems by calling the function with a wrong set of parameters. For example if the calculations inside the function assume that some passed variable `x` is a vector and it is infact a dataframe, some problems will arise.

There is a special behavior of the assignment operator `=` when it is invoked inside the argument string passed to a function. Specifically, if we invoke `function(y = 1:3)`, then instead of using the vector `(1,2,3)` as the first argument to this function, it looks in the function definition for an argument named `y` and assigns the vector to that argument. Thus,

```
> dot(x = 3:5)
[1] 50
> dot(y = 3:5)
Problem in dot(y = 3:5):  argument "x" is missing with no default
Use traceback() to see the call stack
```

This behavior is quite useful when we are dealing with functions potentially taking many arguments. The `printgraph` function, for example, takes 11 arguments, all of which are optional in the sense that they all have default values. The second of these arguments is `print`, which if set to False will cause the display to print just to a PostScript file, not to the printer as well. We can invoke this behavior with `printgraph(print = F)`.

3.2 Defining functions: commenting, working with files

When we begin working with functions more than a few lines in length, retyping the function every time as we work through the debugging process can get quite tedious. It is thus much easier to work on the body of the function by using a text editor to edit a file and then loading the contents of the file. Note that if you are using `emacs`, you can open multiple buffers in which to work on your functions (similar to many other Windows based word processors). The loading of the file is accomplished using the `source` command. We'll illustrate this by defining a function to find the slope and intercept of the least-squares regression line and we want this to appear in some nice form. If we're going to do this repeatedly, we'd like to write a function that takes the necessary arguments (in this case, the vectors `x` and `y`) and returns the desired numbers (here, `b0` and `b1`). A second feature which we will use here is the ability to comment our code; a very good idea for functions more than a few lines in length! The comment character in S is the number sign, #. Anything appearing on a line after the occurrence of a comment character is ignored.

In working with files containing computer code, it is often useful to append a suffix to the filename indicating what type of code the file contains. Thus, `filename.c` denotes a C source file, `filename.f` denotes Fortran, and `filename.ss` denotes Scheme. While it is tempting to use `.s` as the suffix for `S` code files, this suffix has been in use for many years denoting Assembler code. Thus, a loose tradition has developed of using a `.q` suffix for S code. We will adhere to this here.
For now, we will store our function in the file “lscoef.q”.

```
# LSCOEF(X,Y) - finds the coefficients of the least-squares
# regression line fitting a univariate vector Y in terms
# of a univariate vector X. The function returns a vector
# of (slope, intercept).
lscoef <- function(x,y){
    slope = sum((x-mean(x))*y)/sum((x-mean(x))^ 2);
    intercept = mean(y) - slope*mean(x);
    return(c(intercept,slope));
}
```

At the very beginning of the file, we have included a commented statement saying what the function is designed to do. This is a useful tool, since the person most likely to need these comments is you. Even if you never plan to use the code again it is useful to make comments because you never know who might be using it next.

```
> x = 1:10;
> y = 3 + 1.5*x + 2*rnorm(10);
> source("lscoef.q")
> lscoef(x,y)
[1] 3.080144 1.567123
```

Note the quotes around the name of the file! These are needed. You may use single or double quotes; `source('lscoef.q')` would yield the same result. The `source` function reads the specified file in line by line, treating each line as if it were typed at the command prompt. We could have included the definitions of `x` and `y` in the file as well. Unlike MATLAB or Java, there is no restriction on the how the name of a file should relate to its contents. We could have called the above file "junk.q" and the results would have been the same. This is the way that I most often use S, actually; I set up my functions in files and source them into memory, and then I set up another file describing how I want to invoke the functions (these are mostly simulations).

4 Matrices

Quite often in statistics, and particularly in linear models, it is more convenient to work with matrices of data rather than vectors. We may have multiple X values (predictors) associated with a given Y value, for example, and the X values for a given Y could be stored as a row in the matrix. Similarly, matrices can be useful for the tabular arrangement of data such as the homework grades in a class – each row an individual, each column a given homework. Operations on rows and columns make sense in this context. There are four commands which are very useful for creating matrices of data: `matrix, cbind,` and `rbind`.

```
> x <- 1:6
> y <- 7:12
> z <- matrix(x,2,3) # matrix(data,nrows,ncolumns)
> z
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
> z <- matrix(x,2,3,byrow=T) # load data by rows, not by columns
> z
     [,1] [,2] [,3]
```

```
[1,]    1    2    3
[2,]    4    5    6
> z <- cbind(x,y) # bind x and y together as columns of z
> z
     x  y
[1,] 1  7
[2,] 2  8
[3,] 3  9
[4,] 4 10
[5,] 5 11
[6,] 6 12
> z <- rbind(x,y) # bind x and y together as rows of z
> z
  [,1] [,2] [,3] [,4] [,5] [,6]
x    1    2    3    4    5    6
y    7    8    9   10   11   12
> u <- rbind(z,x) # add another row to z
> u
  [,1] [,2] [,3] [,4] [,5] [,6]
x    1    2    3    4    5    6
y    7    8    9   10   11   12
x    1    2    3    4    5    6
> u <- cbind(z,x) # try to add another column; won't work
Warning messages:
  Number of rows of result is not a multiple of vector length (arg 2) in:
cbind(z, x)
```

Note that when you convert a vector of data to a matrix using the `matrix` function, the default is to load the data by columns, filling the first column first, then the second, and so on. This default can be overridden by explicitly using the `byrow = T` option to `matrix`.

There are a number of arithmetical operations that can be performed on matrices; we list some of the basics below (a more complete list is given at the end).

```
> x <- matrix(1:6,2,3,byrow = T)
> y <- 7:12
> x
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
> t(x) # matrix transpose
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
> x * x # elementwise multiplication
     [,1] [,2] [,3]
[1,]    1    4    9
[2,]   16   25   36
```

```
> x * t(x) # not automatically matrix multiplication!
Problem in x * t(x): Dimension attributes do not match
Use traceback() to see the call stack
> x \%*\% x # matrix multiplication
Problem in "%*%.default"(x, x): Number of columns of x should be the same
as number of rows of y
Use traceback() to see the call stack
> x %*% t(x)
     [,1] [,2]
[1,]   14   32
[2,]   32   77
> t(x) %*% x
     [,1] [,2] [,3]
[1,]   17   22   27
[2,]   22   29   36
[3,]   27   36   45
> x + y # matrix to vector, add, then refit - note bycol for y
     [,1] [,2] [,3]
[1,]    8   11   14
[2,]   12   15   18
> x * y # similar elementwise multiplication
     [,1] [,2] [,3]
[1,]    7   18   33
[2,]   32   50   72
> x[,1] # first column of x
[1] 1 4
> x[1,] # first row of x
[1] 1 2 3
> x[1:2,1:2] # square submatrix of x
     [,1] [,2]
[1,]    1    2
[2,]    4    5
> solve(x[1:2,1:2]) # matrix inverse
          [,1]       [,2]
[1,] -1.666667  0.6666667
[2,]  1.333333 -0.3333333
> x + 5 # add 5 to all elements
     [,1] [,2] [,3]
[1,]    6    7    8
[2,]    9   10   11
> x + c(1,10) # vector replication to length of x, then added
     [,1] [,2] [,3]
[1,]    2    3    4
[2,]   14   15   16
```

5 Conditionals and Loops

Just the basics here: as with most other programming languages, S allows for `for`, `while` and `if` structures. These are implemented as follows:

```
> x
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
> for(i in 1:6){ # one way of invoking a for loop
+ x[i] <- x[i] + i;
+ }
> x
     [,1] [,2] [,3]
[1,]    2    5    8
[2,]    6    9   12
> for(i in 1:length(x)){ # an equivalent way in this case
+ x[i] <- x[i] - i;
+ }
> y <- x # default allocation of size, length, etc for y
> for(i in 1:length(x)){
+ if(x[i] < 3) # first conditional if
+ y[i] <- x[i] - 1;
+ else if(x[i] > 4) # next conditional else if
+ y[i] <- x[i] + 1;
+ else # last condtional else
+ y[i] <- x[i]
+ }
> y
     [,1] [,2] [,3]
[1,]    0    1    3
[2,]    4    6    7
> x
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
> i = 5;
> while(i > 0){ # conditional while loop
+ y[i] <- x[i] - 1;
+ i <- i - 1;
+ }
> y
     [,1] [,2] [,3]
[1,]    0    1    2
[2,]    3    4    7
```

Some notes about looping in S. 1) Looping in S is *very slow.* If you can find some way to vectorize your operation instead of invoking a loop, it is almost always to your advantage to do so. 2) Things outside the loop are updated only when the loop is completed. Thus, if you are trying to put together a big simulation that requires a big loop, and you have to hit control-C after only a few hours, there will be nothing stored. 3) Nested loops in S are even slower than regular loops. 4) Avoid looping in S.

One very useful command that avoids some looping is `apply`, which will apply a specified function to all of the rows or columns of a matrix, as directed.

```
> apply(x,1,mean) # means of rows of x
[1] 2 5
> apply(x,2,mean) # means of columns of x
[1] 2.5 3.5 4.5
```

6 Cleanup

When you are done using S-Plus, it is a good idea to clean up after yourself. S assumes that when you leave a session, you will be coming back to it later, so any vectors, functions, or whatnot that you may have defined in your current session will be saved in the .Data file and will be there the next time that you invoke S. This may be a good thing, depending on your working style, or it may not. One thing it can lead to, however, is a situation where the vector `x`, which was important to you a few weeks ago, is still there but now you can't remember what it does and it is taking up space. Doing some cleanup as you go is a pretty good idea.

The first step in cleaning up is finding out what you've got. This is accomplished by invoking `ls()`.

```
> ls()
[1] ".Last.value" ".Random.seed" "dot" "evec" "last.dump"
[6] "lscoef" "sigma" "std.dev" "t.stat" "t.test.p"
[11] "x" "y"
```

Most of the stuff here is stuff that I defined during the course of putting this file together - the functions `dot, lscoef` and the vectors or constants `evec, sigma, x, y`. The functions `std.dev, t.stat, t.test.p` are from an earlier session (see if you can guess what they do) but they're not important. There are three other items listed, and these were generated automatically by S as I went along: `.Last.value, .Random.seed, last.dump`. The vector `.Last.value` is a useful safeguard in case you generated a result and wanted to assign it but forgot to. For example,

```
> 3 + 4
[1] 7
> x <- .Last.value
> x
[1] 7
```

Thus, while we could clean this up, it will reappear the very first time we do anything the next session around, so its not worth it (unless your last operation returned a HUGE file. The vector `.Random.seed` is also innocuous. S-Plus uses a pseudorandom number generator requiring a numerical seed value, which is updated every time another pseudorandom variate is generated. Leave it alone, and your random numbers won't overlap in weird ways. Finally, `last.dump` stores some of the diagnostics from the last time you did something in S-Plus that generated an error message, as it tries to tell you how not to do it again. I tend to get rid of this one, but I know I'll see it again (sigh). To get rid of the stuff we don't want,

```
> rm(dot,evec,last.dump,lscoef,sigma,std.dev,t.stat,t.test.p,x,y)
> ls()
[1] ".Last.value" ".Random.seed"
> q()
```

Invoking `q()` quits the session. Just typing `q` will not do it; it wants to know that a function is being invoked so the parentheses are necessary.

There is one other file created by S-Plus, and this is a "stealth file" so you might not notice it for a while. In your .Data directory, S creates a .Audit file that tracks the commands that you issue to S during your working sessions. This file can be read from the UNIX shell using `more .Data/.Audit` (ie, it has readable text) and can remind you what you were doing the last session or two. The problem is that if you work with S-Plus regularly, this file can grow to be quite large without your being aware of it, so every once in a while you need to explicitly remove it. This cannot be done from inside S; you need to escape to UNIX and `rm .Data/.Audit` to get rid of it.

Written by Blair Christian
Modified by Viacheslav Fofanov