

A Guide for the Unwilling S User

Patrick Burns*

23rd February 2003

Introduction

Two versions of the S language are available—a free version called R, and a commercial version called S-PLUS. Though there are differences, the two versions are largely compatible. Here we use “S” to mean either.

S is an interactive language. S prompts you for input, you type something to S (followed by a “return”), and S responds with an answer. The usual prompt from S is “>”. Examples show the prompt, you type what follows.

To help visualize how S works, reference is made to spreadsheets. If you are not familiar with spreadsheets, just envision a grid of rows and columns.

Starting and Quitting S

How you start depends on the version of S and the operating system. Quit with:

```
> q()
```

In response R asks if you want to save your work. S-PLUS always saves.

The Look and Feel of Objects

Almost everything in S is an object. S objects generally hold a collection of items rather than just a single value. The basic objects can be imagined to be linear (like a single column in a spreadsheet) or rectangular (like a selection of rows and columns in a spreadsheet). Some objects contain only one type of data, others can have a mix of types. Table 1 illustrates.

- The most common objects are *vectors*. Vectors contain numbers, character strings, or logicals (TRUE or FALSE). Only one type is in any particular vector. Think of a vector as a single column in a spreadsheet with a single type of entry.

*The author thanks Peter Flom and Daniel Hoppe for their comments.

Table 1: Basic Objects in S

	linear	rectangular
all the same type	vector	matrix
mixed type	list	data frame

- A *matrix* also has a single type of entry, but has rows and columns. S distinguishes between a vector and a matrix that has one column.
- A *data frame* looks like a matrix, but may have different types in different columns. There is only one type within each column of a data frame.
- A *list* contains components which can be any sort of object including another list. The spreadsheet analogy breaks down, but perhaps closest is to think of a list as a collection of spreadsheets.

Vectors and lists can have *names*—each element or component gets a name. The rectangular objects have *dimnames*—names for each row and each column. Data frames must have dimnames, matrices may or may not.

A *factor* is a division into categories. For example, a factor might have 10 elements that are each “red”, “blue” or “green”. The three colors are the *levels* of the factor. Data frames almost always coerce a character column to a factor.

Naming an Object

You need to name objects that you want to save. One name identifies a whole object—“names” and “dimnames” just specify pieces of objects. The rules are:

- Names are combinations of letters, digits and the period (“.”).
- The underscore (“_”) is NOT allowed.
- Names are case-sensitive, so “ab” and “Ab” are two different names.
- At least one letter must appear before a digit does.
- Some names are reserved, like: “return”, “break”, “if”, “TRUE”, “FALSE”.

A special case is “F” and “T”—alternates for “FALSE” and “TRUE”. In S-PLUS “F” and “T” are reserved. In R you can use them, but it’s a bad idea.

To reduce confusion it is best to avoid using names that are the names of S functions that you use. Common cases where this occurs are “c”, “t” and “q”.

The way to name objects is to give the name, then the two characters “<-” (best surrounded by spaces), then the command that creates the object:

```
> my.pi <- 3.14
```

Most versions allow the alternate command:

```
> my.pi = 3.14
```

Reading from a File

One common way to create objects is to read in data from a file. The “read.table” function puts the contents of a file into a data frame. If your file is tab-separated, then the command would look something like:

```
> df1 <- read.table("some_file.txt", sep="\t")
```

Additional arguments can adapt the command to what is in the file.

Putting Things Together

The “c” function combines elements into a vector:

```
> num.vec <- c(12.34, 56. 8, 90.12, 34.56)
```

If you just type the name of an object, then S will print the object:

```
> num.vec
[1] 12.34 56. 8 90.12 34.56
```

(The “[1]” starting the response says that line begins with the first element.)

The “c” function is not restricted to numbers:

```
> char.vec <- c("Xi'an", "Temir", "Agra", "Krue",
+ "Arica", "Bilma")
```

You need not give all of a command on a single line. Since there was no “)” to match the opening “(“ on the first line, S knows the command is not yet complete and gives its continuation prompt “+”.

With the “name=value” form, the “c” function can put names on the vector:

```
> logic.vec <- c(Brazil=FALSE, "China" = TRUE, Niger=TRUE,
+ "United States"=FALSE)
> logic.vec
      Brazil      China      Niger United States
      FALSE      TRUE      TRUE      FALSE
```

The only name required to be in quotes is “United States” because of the space.

Names can also be put on after the fact:

```
> names(num.vec) <- c("first", "second", "third", "fourth")
> num.vec
first second third fourth
12.34 56. 8 90.12 34.56
```

The “:” operator produces a sequence of numbers:

```
> 1:4
[1] 1 2 3 4
```

One way of creating matrices is to bind vectors together. The “rbind” function treats the vectors as rows:

```
> num.mat <- rbind(num.vec, 1:4)
> num.mat
      first second third fourth
[1,] 12.34  56. 8 90.12  34.56
[2,]  1.00   2.00  3.00   4.00
```

The names from “num.vec” are used as column names. There are no row names.

Lists are created with “list”—almost always with the “name=value” form:

```
> list1 <- list(num=num.vec, char=char.vec, logic=logic.vec)
```

By now we have created a number of objects. To view your objects, do:

```
> objects()
[1] "char.vec" "df1"      "list1"    "logic.vec"
[5] "my.pi"    "num.mat"  "num.vec"
```

Tearing Things Apart

Extracting pieces of objects is called *subscripting*. Most subscripting is done with square brackets. There are four common ways of using square brackets.

- Positive numbers—select the index numbers that you want.

```
> char.vec[c(3:5, 1)]
[1] "Agra" "Krue" "Arica" "Xi'an"
```

- Negative numbers—state the index numbers that you don’t want.

```
> num.vec[-3]
      first second fourth
12.34  56. 8  34.56
```

- Characters—select the names that you want. (There needs to be names.)

```
> num.vec[c("first", "fourth")]
      first fourth
12.34  34.56
```

- Logicals—select the locations that you want.

```
> num.vec[num.vec > 50]
second third
56.8 90.12
> logic.vec[num.vec == 34.56]
United States
FALSE
```

Testing equality with numbers needs to be done with care—the inexactness of computed numbers can cause equality not to hold exactly.

It is possible to make assignments into part of an object:

```
> num.vec2 <- num.vec
> num.vec2[3] <- -93
> num.vec2
first second third fourth
12.34 56.8 -93 .00 34.56
```

In matrices and data frames the rows and columns are subscripted separately:

```
> num.mat[2:1, logic.vec]
second third
[1,] 2.00 3.00
[2,] 56.8 90.12
```

Leave a blank in a dimension to indicate that you want all the rows or columns:

```
> num.mat[, c("second", "fourth")]
second fourth
[1,] 56.8 34.56
[2,] 2.00 4.00
```

For matrices and data frames, character subscripts correspond to dimnames.

The “\$” operator is used to extract a component out of a list:

```
> list1$char
[1] "Xi'an" "Temir" "Agra" "Krue" "Arica" "Bilma"
```

Doing Arithmetic

In arithmetic, multiplication and division are done before addition and subtraction:

```
> 2 * 3 + 25 / 5 - 4
[1]
```

Use parentheses to modify the default order of computation:

```
> 2 * (3 + 25) / 5 - 4
[1] .2
```

From the “[1]” we see that S is thinking of vectors. The vectors need not have the same length. Consider the command:

```
> c(3, 5) + -3:2
[1] 0 3 2 5 4
```

We can visualize this in spreadsheet form as in Table 2. The first two columns of this table show the original problem. The next two columns show the expanded form of the problem—the shorter vector is pasted down its column until it is as long as the other vector. Once the expansion is done, the calculation can be done. The answer is shown in the final column of the table.

Table 2: Addition problem

original		expanded		answer
c(3,5)	-3:2	c(3,5)	-3:2	
3	-3	3	-3	0
5	-2	5	-2	3
	-1	3	-1	2
	0	5	0	5
	1	3	1	4
	2	5	2	7

Missing values are denoted by “NA” and propagate through calculations:

```
> c(2, 4, NA, 8, 10) / 2
[1] 1 2 NA 4 5
```

Use “is.na” to test for missing values—the first command asks if the elements of “num.vec” are equal to some specific number which is unknown:

```
> num.vec == NA
first second third fourth
NA NA NA NA
> is.na(num.vec)
first second third fourth
FALSE FALSE FALSE FALSE
```

Feeding Functions

Using S is mostly calling functions. We’ve already seen a few functions: “c”, “read.table”, “objects” for instance. When using a function, you give values

to the arguments of the function. Usually not all of the arguments need to be given—many will have default values.

Let's go back to reading a file:

```
> df1 <- read.table("some_file.txt", sep="\t")
```

This command calls the “read.table” function using two arguments. The first argument, “some_file.txt”, is not named so it must refer to the first argument of the function. The second argument is named by “sep”. Since it is named, it need not be the second argument of the function.

You seldom need to give the full name of an argument. You only need enough of the first part of the name to make it unique among the argument names of the function. For instance, we could have used “se” to specify the “sep” argument, but we couldn't use just “s” because there is another argument to “read.table” that begins with “s”. Special rules apply to functions, like “c”, that have an arbitrary number of arguments.

View a function's arguments and default values with “args”. For example:

```
> args(read.table)
```

Magical Functions

Some functions are magical. Examples are “print”, “plot” and “summary”. What happens depends on what is given. This saves you a lot of bother.

When a data frame is printed, it looks like a matrix. However, the actual structure of a data frame is far from a matrix. When “print” sees a data frame, it hands the work to the function specially built to print data frames.

Magic functions like “print” are said to be *generic*. What they do depends on the *class* of the object passed in. The whole scheme of having magic functions is called object-oriented programming.

Making Pictures

In many cases something useful happens when you “plot” an object. For a single vector, the plot is against the index numbers.

```
> plot(num.vec)
```

Plots need a place to be—the plot is sent to a *graphics device*. Generally if a graphics device is not active, S will start an appropriate one. You can control which device is used, for instance “postscript” creates PostScript files.

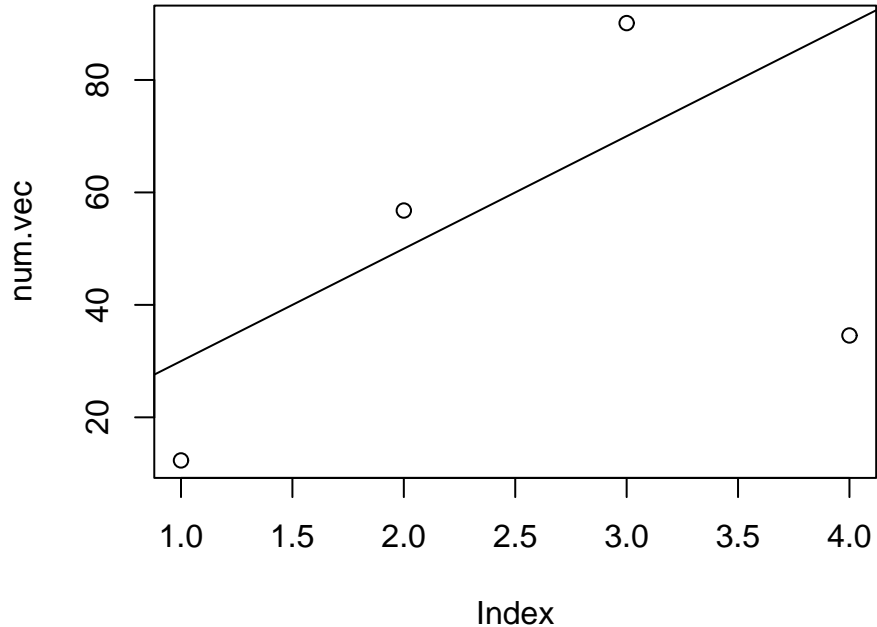
Graphics are modular—you can add to them:

```
> abline(10, 20)
```

This adds a line with intercept 10 and slope 20 to the plot, yielding Figure 1.

Other useful graphics functions include “boxplot”, “hist”, “barplot”. Graphical details are controlled by “par”.

Figure 1: An example plot



Going Farther

You can learn about specific S functions by typing a question mark followed by the name of the function. For example:

```
> ?objects
```

A more cumbersome way of doing exactly the same thing is:

```
> help(objects)
```

There are a variety of books and tutorials for learning S. One example is “R for Beginners” by Emmanuel Paradis which can be found in the contributed documentation section of the R Project website <http://www.r-project.org/>. “R for Beginners” is available in a few languages other than English. The R website also has a FAQ which covers a number of questions. The support section of <http://www.insightful.com/> has a list of books about S or using S.

Both R and S-PLUS have mailing lists for help and discussion. The R Project website has details about R-help. The <http://www.biostat.wustl.edu/S-news/> site gives information about S-news. Both lists have archives you can search to see if your question has already been asked and answered.

More words associated with S are explained in the S Poetry glossary which can be found on <http://www.burns-stat.com/>. The document you are reading can be found in the Tutorials section of the same website.