Programming in S-PLUS*

Aidan Palmer

August 23, 1996

1 Introduction

This handout is intended as an introduction to programming in S-PLUS The reader is assumed to be familiar with S-PLUS and to be familiar with elemenaty programming concepts like looping and branching.

2 Writing Functions

S-PLUS comes with many functions built in, such as `mean`, `log`, `lm` and others. S-PLUS also allows you to create and run your own functions.

To create a function, you must tell S-PLUS how many arguments the function takes, and what it does. The following series of commands would create a function of two variables and store it as `my.function`. When a function has finished executing, it returns the value of its last line, in this case the value of `value.to.return`.

```
my.function <- function(arg1, arg2)
{
...
value.to.return
}
```

The curly brackets `{ and }` are used to indicate a batch of commands which should be executed together. The `function` command expects to evaluate only one thing, so to evaluate multiple commands, enclose them in curly brackets.

S-PLUS, for some reason, does not have a standard deviation function. Why not write one?

```
> stdev <- function(x)
+ {
+ sqrt(var(x))
+ }
```

(Recall that when an incomplete command has been entered, S-PLUS provides a `+` prompt until the command is completed.) The curly brackets are not necessary for this function, since it consists of only one command, but they don't hurt.

To demonstrate how the function works, make up some vector and try the `stdev` function.

```
> z <- c(1, 2, .45, -1.2, 2.3)
> stdev(z)
[1] 1.395708
```

*In writing this handout I was helped greatly by the S-PLUS Help Archive maintained by the University of Toronto Department of Statistics

Functions are a type of object in S-PLUS, like vectors and data frames. This means they can be manipulated like the other objects.

```
> is.function(stdev)
[1] T
> my.stdev <- stdev
```

Assignments within a function have only "local" scope. That is, they do not affect the value of any external objects. Consider the following example, in which the value of `test` is changed within a function, but `test` has its original value when the function is through.

The function `changer` takes no arguments and changes the value of `test` to `"changed"`. The function then returns the value of `test`.

```
> test <- "unchanged"
> changer <- function()
+ {
+ test <- "changed"
+ test
+ }
> changer()
[1] "changed"
> test
[1] "unchanged"
```

Since `changer` includes two commands, we need the curly brackets at the beginning and at the end. Also note that the last line of `changer`, `test` is what the function returns when it is through executing.

Sometimes local scope will seem like a good thing to you, since there is no danger of a function doing any unexpected damage to your data. On the other hand, it will sometimes make editing data more complicated than you would like.

If you did actually want to use `changer` to change the value of `test`, you could do this:

```
> test
[1] "unchanged"
> test <- changer()
> test
[1] "changed"
```

For complicated functions, it is helpful to include comments explaining what the function does. Anything on a line after a # symbol will be ignored in executing the function, but will show up when the function is displayed. This means that a line of comments should be preceded by `#`.

```
> stdev <- function(x)
+ {
+ # This function calculates standard deviation
+ sqrt(var(x))
+ }
> stdev
function(x)
{
# This function calculates standard deviation
        sqrt(var(x))
}
```

3 Editing Functions

You do not need to retype a function every time you want to make a change. S-PLUS provides a function, `ed` allowing you to edit functions in a special window. The function `ed` uses its own editor, but you can specify that you would rather use emacs.

```
> stdev <- ed(stdev, editor="emacs")
```

The above command creates an emacs window in which the current definition of `stdev` can be edited. Once editing is completed, the new version is saved as `stdev`.

If you don't want to type `editor="emacs"` every time, create your own function called `edit`, which invokes `ed` and uses emacs.

Question: How?

4 Loops and Logic

At this point you should be able to write straightforward functions that apply a series of command to several arguments. However, what if you want to do something more complicated?

To do something a set number of times, use a `for` loop, like this: `for (`*index* `in` *range*`) { }`

The variable used as the index takes on each value listed in the range, in order. For example, the following `for` loop is used to sum the numbers 1 through 10.

```
> total <- 0
> for(i in 1:10)
+ {
+ total <- total+i
+ }
> total
[1] 55
```

This also works for more complicated ranges:

```
> total <- 0
> for(i in c(1,2,4,5,9))
+ total <- total+i
> total
[1] 21
```

The following function demonstrates a more useful application of a `for` loop. The function `stretch`, described below, takes a vector and stretches it out, by repeating each element of the vector the same number of times. (The body of the function creates a blank vector, and then appends the copies of the original elements). This is sometimes useful in reformatting data. The guts of this function uses the function `rep`, which generates a vector conataining the first argument repeated the number of times specified by the second argument.

```
> stretch <- function(vec, num)
+ {
+         output <- c()
+         for(i in 1:length(vec))
+                 output <- append(output, rep(vec[i], num))
+ }
> stretch(c("big bird", "oscar", "snuffleupagus"),4)
 [1] "big bird"      "big bird"      "big bird"      "big bird"
 [5] "oscar"         "oscar"         "oscar"         "oscar"
 [9] "snuffleupagus" "snuffleupagus" "snuffleupagus" "snuffleupagus"
```

Note that `rep` can be used to lengthen the data in a different way:

```
> rep(c("big bird", "oscar", "snuffleupagus"),4)
 [1] "big bird"      "oscar"         "snuffleupagus" "big bird"
 [5] "oscar"         "snuffleupagus" "big bird"      "oscar"
 [9] "snuffleupagus" "big bird"      "oscar"         "snuffleupagus"
```

An if statement looks like `if (`*expression*`) {   } else {   }`

The expression can be any logical statement, using `<`,`>`, `==`, `&` (and), `|` (or), etc. If the expression evaluates to true, the part withn the first set of brackets will be executed. If it is false, the `else` section, if one exists (it is optional) will be executed.

```
> x <- 1
> if(x < 1) {"x is less than one"} else {"x is greater than or equal to one"}
[1] "x is greater than or equal to one"
```

A `while` loop can do everything a `for` loop can do, but it can handle more general loops as well. The format is `while (`*expression*` ) {   }`

The following function will calculate the largest integer power (greater than zero) of a positive number `x` which is less than `y`. The two `if` statements ensure that the value of `x` will work. The next two instructions initialize `i` and `total`. The `while` loop is the guts of this function.

```
> max.power <- function(x, y)
+ {
+         if(x < y & x > 0)
+                 if(x > 1) {
+                         i <- 1
+                         total <- x
+                         while(total * x < y) {
+                                 i <- i + 1
+                                 total <- total * x
+                         }
+                         i
+                 }
+                 else "Infinity"
+         else NULL
+ }
```

The two `if` statements ensure that x and y make sense in this context. The `while` statement keeps multiplying by `x` and counting how many times it has done this. In the end, `i`, `"Infinity"` or `NULL` is returned.

```
> max.power(2, 100)
[1] 6
> max.power(2, 1)
NULL
> max.power(.5, 100)
[1] "Infinity"
> max.power(-3, 100)
NULL
```

> **Question:** How would you re-write `max.power` to return more informative messages than “NULL” for cases where the first argument is larger than the second, or the first argument is negative?

5 Apply

When you have time and memory constraints, the function `apply` is often more useful than a loop. Consider the `iris` data set (included with S-PLUS) which consists of 50 observations of four characteristics of three types of iris. To find the mean observation for each characteristic, one possibility is:

```
> get.iris.mean <- function(x)
+ {
+ report <- rep(0,4)
+ for(i in 1:4)
+ report[i] <- mean(iris[,i,])
+ report
+ }
> get.iris.mean(iris)
[1] 5.843333 3.057333 3.758000 1.199333
```

A cleaner way, which does not involve any looping, is:

```
> apply(iris, 2, mean)
 Sepal L. Sepal W. Petal L. Petal W.
 5.843333 3.057333    3.758 1.199333
```

The above says "with the matrix `iris`, along margin 2 apply the function mean".

To find the mean for each characteristic (margin 2) for each type of iris, use:

```
> apply(iris, c(2,3), mean)
         Setosa Versicolor Virginica
Sepal L.  5.006      5.936     6.588
Sepal W.  3.428      2.770     2.974
Petal L.  1.462      4.260     5.552
Petal W.  0.246      1.326     2.026
```

6 Bulletproofing

You may want to put some error handling in your functions. S-PLUS has a few functions which do this: `stop` and `warning`. The function `stop` will return an error and terminate the function, while `warning` will return an error message but continue executing the function.

The following standard deviation function works the same as before but returns an error if its argument is not a vector (the exclamation point before `is.vector` means "negation", so the error message appears "if x is not a vector").

```
> stdev <- function(x)
+ {
+ # stdev with error handling
+ if(!is.vector(x))
+ stop("The argument to stdev must be a vector")
+ else sqrt(var(x))
+ }
```

As an example, consider the vector `z` which was used to test the standard deviation function before, and `z` after it is converted to a matrix.

```
> stdev(z)
[1] 1.395708
> z <- as.matrix(z)
> stdev(z)
Error in stdev(z): The argument to stdev must be a vector
Dumped
```

S-PLUS returns the error message verbatim (so when you write error messages, make sure that they are informative).

7 Debugging Functions

S-PLUS has a few functions which are helpful for figuring out where errors are ocurring. The function `inspect` is an interactive debugger, which allows you to step through a function and check the values of the variables at each point. Usually, this will be sufficient to identify a bug, but `inspect` has more advanced features as well.

Type `inspect(max.power(2,9))` to trace through a call to `max.power` with arguments 2 and 9. The `inspect` prompt, `d>` will appear. Type `help` to see a list of commands, and `objects` to see a list of variables defined in the function.

The main commands you will need are `do` to advance one command of the program, `step` to advance one line of a loop of a program, and `eval` to evaluate a variable. You can advance multiple lines by typing `do` followed by the number of lines, and a similar thing works for `step`. The difference between `do` and `step` is that `do` treats a block of commands (for instance, in a loop) as a single command, while `step` looks at each command separately.

The command `resume` ends the debugger after running through the rest of the program, and `quit` ends the debugger without executing any more.

8 An Example

Note: in this section, numerous mistakes are made, for illustrative purposes.

Consider a function for breaking a vector into any number of quantiles. If the data look like `c(.42,.15,.25,.27,.35,.38,.41,.1)` for instance, the two-quantile breakdown would be `c(2,1,1,1,2,2,2,1)` and the four-quantile version would be `c(4,1,2,2,3,3,4,1)`.

S-PLUS has a function, `quantile` which takes a vector of data and a vector of probability breakpoints (`c(0,.5,1)` for two quantiles, `c(0,.25,.5,.75,1)` for four, etc.) and returns the breakpoints for the quantiles of data. This will be the guts of the function.

The function will look something like this:

```
by.quantiles <- function(x, n)
{
```

Generate the vector of probability breakpoints for n quantiles.
Use `quantile` *to generate the actual breakpoints.*
Use these breakpoints to classify the data, x, into n quantiles.
Return the result.

```
}
```

The first step is to figure out how to generate the probability breakpoints. We know that this will be some vector, starting with 0 and ending with 1. The vector to split the data in half conatained three elements (0, .5, 1) and the vector to split the data in four parts took five elements (0, .25, .5, .75, 1) so we can figure that the vector for splitting the data into `n` parts will require `n+1` elements. We can use the `rep` function to create a framework for this vector. It is probably best to try out each part of the function separately, until we know that all the pieces will fit together.

```
> by.quantiles <- function(n,x)
+ {
+ pbreaks <- rep(0, n+1)
+ pbreaks[n+1] <- 1
+ pbreaks
+ }
> by.quantiles(c(1,2,3,4),2)
Error in rep.int: rep() only defined for length(times)==1 or length(x): rep.int(
1:xlen, times)
        . . .
Dumped
```

Woops, `x` should be the first argument, `n` the second. After that correction, the test run produces "`[1] 0 0 1`" as output, which looks right so far. We need to figure a way to calculate the middle numbers in the vector. One way to do this could be a for-loop. We know the first and last numbers in the vector, and want to fill in the middle numbers, one at a time. In the example of two quantiles the vector counted by halves, (0, .5, 1) and in the example of four quantiles the vector counted by quarters, (0, .25, .5, .75, 1). In the case of `n` quantiles, the vector should count by 1/n.

Use the `ed` or `edit` function to change `by.quantiles` to the following form:

```
function(x, n)
{
        pbreaks <- rep(0, n + 1)
        pbreaks[n + 1] <- 1
        for(i in 2:(n-1))
                pbreaks[i] <- 1/n
        pbreaks
}
```

Now let's try this out:

```
> by.quantiles(c(1,2,3,4),2)
[1] 0.5 0.5 1.0
```

Hmm, that doesn't look right, the first one should be 0. Let's try some different input to get a better idea of what is going wrong.

```
> by.quantiles(c(1,2,3,4),4)
[1] 0.00 0.25 0.25 0.00 1.00
```

This shows a couple of problems: the 0 is showing up next-to-last, and the numbers are not increasing. The second problem is coming from the `pbreaks[i] <- 1/n` line. We want to get `i` somewhere on the right side of that equation. The first problem is because we used `n-1` instead of `n` in the for-loop. The vector has `n+1` elements, so the next-to-last element is element `n`, not `n-1`.

Fixing the range of the for-loop should solve the first problem, and let's try `i/n` instead of `1/n` to solve the second problem.

```
> by.quantiles(c(1,2,3,4),2)
[1] 0 1 1
> by.quantiles(c(1,2,3,4),4)
[1] 0.00 0.50 0.75 1.00 1.00
```

Woops, it should be `(i-1)/n`. After fixing that bug, this part of the function does what it is supposed to.

```
> by.quantiles
function(x, n)
{
        pbreaks <- rep(0, n + 1)
        pbreaks[n + 1] <- 1
        for(i in 2:n)
                pbreaks[i] <- (i - 1)/n
        pbreaks
}
> by.quantiles(c(1,2,3,4),4)
[1] 0.00 0.25 0.50 0.75 1.00
```

The next step is to use the `quantile` function to calculate the breaks between the quantiles in the data. Type `help(quantile)` to find out how to use the this function. To prevent having to type in a vector of data every time we test the program, store a vector of data as `sample`.

```
> sample <- c(.42,.15,.25,.27,.35,.38,.41,.1)
```

Now, let's put in the `quantile` function and test it out on `sample`. The values returned for `n`=4 should be something like (.1,.2,.3,.4,.5).

```
> by.quantiles
function(x, n)
{
        pbreaks <- rep(0, n + 1)
        pbreaks[n + 1] <- 1
        for(i in 2:n)
                pbreaks[i] <- (i - 1)/n
        dbreaks <- quantile(x, probs = pbreaks)
        dbreaks
}
> by.quantiles(sample, 4)
   0%   25%  50%    75% 100%
  0.1 0.225 0.31 0.3875 0.42
```

That looks about right. The next step is to use `dbreaks` to write the data by quantiles. What we want to do is look at each element of data, and if it is between two breakpoints, it is in that quantile.

We can start by generating `n+1` zeroes, and then using a for-loop to fill them in. Actually, we will need two for-loops: one to bring in each element of data, another to check that element against each pair of quantiles. Note that this means that each element will continue to be checked against each quantile, even after it has been placed in the correct quantile. We could write a faster function which stops checking once it finds the correct quantile, but that would be more complicated to write.

The two for-loops (one on `i`, the other on `j`) will contain an if statement, which asks whether the element fits in the `j`'th quantile. If it does, the value `j` is recorded for that element.

```
> by.quantiles <- edit(by.quantiles)
> by.quantiles
function(x, n)
{
        pbreaks <- rep(0, n + 1)
        pbreaks[n + 1] <- 1
        for(i in 2:n)
                pbreaks[i] <- (i - 1)/n
        dbreaks <- quantile(x, probs = pbreaks)
```

```
        data.by.quantiles <- rep(0, n + 1)
        for(i in 1:(n + 1))
                for(j in 1:n)
                        if(x[i] >= dbreaks[j] & x[i] < dbreaks[j + 1])
                                data.by.quantiles[i] <- j
        data.by.quantiles
}
> by.quantiles(sample, 4)
[1] 0 1 2 2 3
```

How come the output is not as long as the original data? Let's use `inspect` to try to figure out what is going wrong inside those for-loops. (Also, what is that zero doing there?)

```
> inspect(by.quantiles(sample, 4))
entering function by.quantiles
stopped in by.quantiles (frame 3), at:
        pbreaks <- rep(0, n + 1)
d> do
stopped in by.quantiles (frame 3), at:
        pbreaks[n + 1] <- 1
d> do
stopped in by.quantiles (frame 3), ahead of loop:
        for(i in 2:n)
                pbreaks[i] <- (i - 1)/n
d> do
stopped in by.quantiles (frame 3), after loop:
        for(i in 2:n)
                pbreaks[i] <- (i - 1)/n
d> do
stopped in by.quantiles (frame 3), at:
        dbreaks <- quantile(x, probs = pbreaks)
d> do
stopped in by.quantiles (frame 3), at:
        data.by.quantiles <- rep(0, n + 1)
d> do
stopped in by.quantiles (frame 3), ahead of loop:
        for(i in 1:(n + 1))
                for(j in 1:n)
                        if(x[i] >= dbreaks[j] & x[i] < dbreaks[j + 1])
        ...
d> eval x
[1] 0.42 0.15 0.25 0.27 0.35 0.38 0.41 0.10
d> eval dbreaks
   0%   25%  50%    75% 100%
  0.1 0.225 0.31 0.3875 0.42
d> eval data.by.quantiles
[1] 0 0 0 0 0
d> quit
```

Aha! The `data.by.quantiles` vector should have the same length as `x`, not the same length as `dbreaks`. This means `i` should go from `1` to `length(x)`.

```
> by.quantiles <- edit(by.quantiles)
> by.quantiles
```

```
function(x, n)
{
        pbreaks <- rep(0, n + 1)
        pbreaks[n + 1] <- 1
        for(i in 2:n)
                pbreaks[i] <- (i - 1)/n
        dbreaks <- quantile(x, probs = pbreaks)
        data.by.quantiles <- rep(0, length(x))
        for(i in 1:length(x))
                for(j in 1:n)
                        if(x[i] >= dbreaks[j] & x[i] < dbreaks[j + 1])
                                 data.by.quantiles[i] <- j
        data.by.quantiles
}
```

This looks better, but still doesn't work.

```
> by.quantiles(sample, 4)
[1] 0 1 2 2 3 3 0 1
```

Let's use **inspect** again and see what's going wrong. Remember, the output should be (4, 1 2, 2, 3, 3, 4, 1), so how come the function is generating 0's instead of 4's? Or, since the output starts with 0's, why is it leaving them as 0's instead of generating 4's?

```
> inspect(by.quantiles(sample, 4))
entering function by.quantiles
stopped in by.quantiles (frame 3), at:
        pbreaks <- rep(0, n + 1)
d> do 6
stopped in by.quantiles (frame 3), ahead of loop:
        for(i in 1:length(x))
                for(j in 1:(n - 1))
                        if(x[i] >= dbreaks[j] & x[i] < dbreaks[j + 1])
        ...
d> step 5
stopped in by.quantiles (frame 3), at:
        if(x[i] >= dbreaks[j] & x[i] < dbreaks[j + 1])
                data.by.quantiles[i] <- j
d> eval j
[1] 4
d> eval data.by.quantiles[1]
[1] 0
```

At this point S-PLUS has checked the first element (.42) against the fourth quantile, but has recorded a 0 meaning that it does not fit. Why not?

```
d> eval x[i] >= dbreaks[j]
  75%
    T
d> eval x[i] < dbreaks[j+1]
 100%
    F
```

Hmm, the first element is greater than or equal to the 75th percentile, but not less than the 100th percentile. Why not?

```
d> eval x[i]
[1] 0.42
d> eval dbreaks[j+1]
 100%
 0.42
```

Aha! The test should be whether `x[i]` is less than or equal to `dbreaks[j+1]` instead of strictly less than. After this correction, the function works as it is supposed to. The working function looks like this:

```
function(x, n)
{
        pbreaks <- rep(0, n + 1)
        pbreaks[n + 1] <- 1
        for(i in 2:n)
                pbreaks[i] <- (i - 1)/n
        dbreaks <- quantile(x, probs = pbreaks)
        data.by.quantiles <- rep(0, length(x))
        for(i in 1:length(x))
                for(j in 1:n)
                        if(x[i] >= dbreaks[j] & x[i] <= dbreaks[j + 1])
                                data.by.quantiles[i] <- j
        data.by.quantiles
}
```