10.0 Help, Editing data, Databases and Frames in `Splus`

Notes based on those of Professor Joe Schafer

Main subjects covered today:

- help.start()
- data.ed()
- Databases
- Frames

10.1 Interactive help with help.start()

Until now, we've been able to get help about a specific functions by typing "help(functionname)". This is useful if you already know the name of the function. But what if you don't? Typing

```
> help.start()
```

will open up a window on the screen that provides interactive help by topic. If you have a particular technique in mind–factor analysis, for example–you can also search for functions that will do it. Try it out.

Depending on your Splus installation, you may need to include the option gui="motif", like this:

```
> help.start(gui="motif")
```

10.2 Interactive editing of data objects

To edit an object interactively, use the `data.ed()` function. For example, suppose you want to change one or more elements of a vector or matrix named y. Type:

```
> data.ed(y)
```

If that doesn't work, try:

```
> data.ed(y,gui="motif")
```

Splus will open up a window that allows you to view and edit the individual elements of y.

10.3 Databases

To really understand how S and Splus work, you need to know something about databases. A database is a directory on the disk in which objects are stored. The most obvious database is your .Data directory, which holds the data objects and functions created by you. But there are other databases as well. For example, there is a database that holds all of the regular S functions like `c()`, `print()`, and so on.

When you type a command, for example

```
> x <- c(1:3,y)
```

Splus will search for the objects in an orderly fashion. In this example there are two objects that need to be found: a function called "c" and a data object called "y". Splus will first search your .Data directory, where it will presumably find “y” but not “c”. Then it will search other databases until it finds the function “c”.

The function `search()` allows you to see the “search list”, a listing of all databases that Splus will search:

```
> search()
[1] ".Data"
[2] "/usr/local/splus-3.3/splus/.Functions"
[3] "/usr/local/splus-3.3/stat/.Functions"
[4] "/usr/local/splus-3.3/s/.Functions"
[5] "/usr/local/splus-3.3/s/.Datasets"
[6] "/usr/local/splus-3.3/stat/.Datasets"
[7] "/usr/local/splus-3.3/splus/.Datasets"
```

This means that Splus first searches the .Data directory; then it searches /usr/local/splus-3.3/splus/.Functions; and so on.

10.4 Masking objects with the same name

Suppose that you create an object with the same name as a pre-existing S or Splus function. For example:

```
> c <- 12.979
```

Later, if you refer to the Splus object with the same name, you will get a warning message:

```
> y <- c(7,4,90.1)
Warning messages:
  Looking for object "c" of mode "function", ignored one of
          mode "numeric"
```

Now we can understand what this warning message means. Splus was looking for a function named “c”. In your .Data directory it encountered an object named “c”, but it was a numeric data object rather than a function. After that, Splus continued searching the databases on the search list until it encountered a function named “c”. Notice that in this example Splus did the right thing. But if you ever get a warning message like that, it would be wise for you to change the name of your object to something other than “c”. For example, you can say:

```
> cc <- c
> rm(c)
```

Because .Data is the first database on the search list, Splus will delete your data object “c” rather than the function “c”. In fact, it would be impossible for you to remove the function “c”, because Splus and the Unix permissions are set up in such a way that you have no permission to do so. So you don’t have to worry about accidentally deleting pre-existing Splus functions.

10.5 More about `objects()`

As we learned last time, the `objects()` function displays a list of all the objects stored in your .Data directory. But it can also display the objects in other databases as well. It has an argument that allows you to refer to the various databases on your search list. The default value of this argument is 1, which says that you want the first database on your search list (.Data). Therefore, typing

```
> objects(1)
```

does the same thing as typing

```
> objects()
```

which display everything in .Data. Typing

```
> objects(2)
```

will display everything contained in the second database in your search list; and so on. Try typing "objects(i)" for various values of i and see what you find.

10.6 Changing the search list

You can attach new databases to the search list using attach("directoryname"). For example, suppose that you have in your home directory a subdirectory called sfunctions, and in sfunctions there is a .Data directory containing some Splus objects (functions, etc.) that you would like to have access to in your current session. Typing

```
> attach(" ~/sfunctions/.Data")
```

will attach ~/sfunctions/.Data to your search list. By default, Splus attaches it in position 2, just below the .Data directory. Now if you type

```
> search()
```

you will see that ~/sfunctions/.Data occupies position 2, and the databases that formerly occupied positions 2,3,... have now shifted down to positions 3,4,... If you type

```
> objects(2)
```

you will see the contents of this newly attached database. You can detach this database by typing

```
> detach(2)
```

which removes the database in position 2 and shifts the databases in positions 3,4,... to 2,3,...

10.7 Frames

Frame is a temporary database created in RAM for the purpose of executing a function. Every time that you call a function, Splus creates a frame to store objects that are local to that function. After the function call is complete, the result is returned and the frame is destroyed.

For example, suppose we have a function that calculates a correlation matrix from a covariance matrix:

```
> cormat <- function(x)
+  m1 <- matrix(diag(x),nrow(x),nrow(x))
+  x*m1*t(m1)
```

Then suppose you call the function like this:

```
> rmat <- cormat(z)
```

Splus creates a frame, attaches it to the search list in a position above the .Data directory. Splus then makes a copy of the object z from .Data and puts it into this frame, giving it the name x. Then it creates m1 from x and puts it into the frame. Finally, Splus calculates x*m1*t(m1), writes the result to rmat in .Data, and destroys the frame.

Notice that if you had an object named “m1” in .Data, that object would be unaffected by the computations that go on within the frame. So when you are writing a function, you don’t have to worry if the name of an object created within the function might match the name of something in your .Data directory. The object m1 in the frame will mask any object in .Data (or any of the other databases, for that matter) having the same name.

Also, suppose that you refer to an object within a function that (a) has not been created within the function and (b) is not an argument to the function. For example, suppose you define a function like this:

```
> dosomething <- function(x)
+ result <- x+y
+ result
```

Notice that y is not created within the function, nor is it an argument to the function. You probably meant to say

```
> dosomething <- function(x,y)
+ result <- x+y
+ result
```

but you forgot to declare y as an argument. What happens? Suppose you call the function like this:

```
> dosomething(9.8)
```

Splus will create a frame containing an object named x whose value is 9.8. When Splus tries to interpret the line “result ¡- x+y” and finds that there is no object named y in the frame, it goes down the search list, searching .Data and all the other databases for something called “y”. If you happen to have an object called “y” in .Data then the function may execute without giving an error message. But if there is nothing called “y” in .Data or in any other database then an error will result.

Generally speaking, it’s bad practice to write a function in which you refer to an object that is neither created locally within that function nor is an argument to the function. It can lead to programming errors that are difficult to detect.

10.8 Errors within functions

Whenever you call a function, Splus opens a frame to perform the necessary computations. What happens when an error occurs? If an error occurs somewhere within the function, then Splus aborts the whole process and destroys the frame. Nothing will change in your .Data directory. There is no such thing as “partial execution” of a function; if even one mistake occurs the whole process is aborted.

The same thing happens when you are running commands from an external file with `source()`. Recall that

```
> source("filename")
```

tells Splus to execute all the commands in the file called “filename”. The `source()` commands opens up a frame. If even one command in the file cannot be performed, the process is aborted, the frame is destroyed, and no changes will be made to anything in .Data.

10.9 Recursion

Loosely speaking, recursion occurs when a function calls itself. Splus supports a limited amount of recursion. For example, suppose you write a recursive algorithm to calculate the factorial of a positive integer:

```
> factorial <- function(x)
+   if(x>1) result <- x*factorial(x-1)
+   else result <- x
+   result
```

What happens when you call this function? It's something like what happens when you hold up a mirror in front of another mirror; you see a long sequence of images within images. Suppose you type:

```
> factorial(10)
```

Splus will first open a frame that contains x=10. Then because 10¿1 it will need to execute "result ¡- x*factorial(x-1)", which involves another call to `factorial()`. As a result it will have to open another frame that contains x=10-1=9. Continuing in this fashion, it will open frames with x=8, x=7, and so on, until it opens a frame containing x=1. At that point, the "else" statement takes effect, destroying the last frame and returning result=1 to the frame containing x=2. Then that frame is destroyed, returning result=2*1=2 to the frame with x=3; then that frame is destroyed, returning result=3*2 to the frame with x=4; and so on.

Recursion is a nifty idea, but it tends to use up large amounts of memory because so many frames need to be created. For example, this factorial function will not work for any argument greater than

50 because it exceeds the default value for the maximum number of frames. So try not to use recursion on a regular basis. (Any computation that can be done recursively can also be done iteratively.)