debugging in R

In the words of Pete Salzman and Norman Matloff, *the principle of confirmation is the essence of debugging*:

```
#Fixing a buggy program is a process of confirming, one by one,
#that the many things you believe to be true about the code actually
#are true. When you find that one of your assumptions is not true,
#you have found a clue to the location (if not the exact nature) of a bug.
```

The idea is to begin small and confirm the behavior of the coding within scripts and objects. Programmers generally agree that code should be written in a **modular** manner. For example, the first-level code should consist of around a dozen lines, containing of mostly calls to functions. The called functions should also be concise and call other functions if necessary. The debugging should be in a **top-down** manner equally. Prior to calling the **debug( )** function in R on an object, run the object to examine the output; whether expected values are returned or not.

Additionally, **Antibugging** approaches can be applied within the syntax. For example, the return of variable **x** should be position, the following code could be inserted:

```
stopifnot(x > 0)
```

Failure to comply with the code will call the **stopifnot( )** function is called:

```
Error: x > 0 is not TRUE
```

Traditional methods of debugging involved insertions of temporary **print( )** statements:

```
x <- y^2 + 3*g(z,2)
cat("x =",x,"\n")
w <- 28
if (w+q > 0) {
   u <- 1
   print("the 'if' was done")
} else {
   v <- 10
   print("the 'else' was done")
}
```

However, inserting temporary **print( )** functions becomes slow and distracting through many iterations.

using debugging facilities in R

The core R debugging facilities consist of the ***Browser***, allowing single steps through the code. The ***Browser*** is called through the **debug( )** function or the **browser( )** function. The debugging facility is specific to functions; for example, function **f( )** is set to debug through **debug(f)**. The ***Browser*** will automatically enter into the beginning of the function until the argument is dropped via **undebug(f)**. Conversely, with long function, the process can be called upon certain lines within a function with the **browser( )** function. To enter a function and remove the set after exiting, the **debugonce( )** function is used effectively.

When the ***browser*** is entered, the command line changes from **>** to **Browse[d] >** with options:

- **n** (for next): executes the next line and then pause. Hitting **ENTER** causes the latter action equally.
- **c** (for continue): like **n**, except that several lines of code may be executed before the next pause. In a loop, this command will result in the remainder of the loop being executed and then pausing upon exit from the loop. In a function but not in a loop, the remainder of the function will be executed before the next pause
- Any R command: still in R's interactive mode and thus can query the value of **x** by simply typing **x**. a variable with the same name as a browser command must explicitly be call something **print( )**

… **where**: prints a stack trace. Displays what sequence of function calls led execution to current location
… **Q**: This quits the browser, returning to R's main interactive mode

The **setBreakpoint( )** function can be called to set certain key locations to bug within the code. Additionally, the **browser** can be called directly in the code by inserting calls to **browser( )** directly within the code. The call to **browser** can be contingent through use of the **expr** argument:

```
> browser(s > 1)          #enters the browser when variable x is greater than 1
> if (s > 1) browser()    #produces the same effect as above
```

A practical example of the above illustration is given a function that has **50** iterations. In order to save time and focus the debugging process, the following code could be used assuming the loop index is **i**:

```
> browser(s > 49)          #enters the browser when variable x is greater than 49
> if (s > 49) browser()    #produces the same effect as above
```

The **setBreakpoint( )** function can be used in the format as follows:

```
> setBreakpoint(filename, linenumber)
```

The above function calls the ***browser*** at the ***linenumber*** passed through as an argument. This is useful through the following example. Assuming the current position in the browser is at **line 12,** within the **x.R** file, and the breakpoint is best set at line **28**, the following function is applied:

```
> setBreakpoint(x.R, 28)
```

The above would require exiting the **browser**, adding a call to **browser( )** at line **28**, and reentering the function if the **setBreakpoint( )** function was not called instead. The **setBreakpoint( )** function works by calling the **trace( )** function. If the **setBreakpoint( )** function had been called on function **g( )**, the breakpoint could be cancelled as follows:

```
> untrace(g)
```

Equally, the **setBreakpoint( )** function can be called whether or not currently in the **debugger**.

tracking in R

The **trace( )** function is flexible with a steep initial learning curve for operating.

```
> trace(f, t)
```

The above instructs R to call function **t( )** each time the function **f( )** is entered. For example, if a breakpoint is to be set at the beginning of function **gy( )**, the following command is applied:

```
> trace(gy, browser)
```

The effect of the above is the same as if the **browser( )** command had been placed in the source code of **gy( )** except considerably more convenient. It is noted that calling the **trace( )** function does **not** alter the source code; instead, the latter alters a *temporary* version of the file maintained by R. The above would subsequently be undone through the following code:

```
> untrace(gy)
```

performaing checks after a crash in R

When R crashes without using the debugger, there is still a debugging tool available after the crash. A ***post-mortem*** can be performed calling the **traceback( )** function. The information regarding the function of the crash will be returned in addition to the call chain of the latter function. More information is obtained if R is set up to **dump frames** in the event of a crash:

```
> options(error = dump.frames)
```

If R is set up as above, the **debugger( )** function can be called after the event of a crash

```
> debugger()
```

The user will then select the level of function calls to inspect regarding the crash in R. For each chosen level, the values of the variables can be examined. After viewing a given level, the **debugger( )** can be returned by selecting the **N** key on the keyboard.

Additionally, R can be set to enter the debugger automatically:

```
> options(error = recover)
```

Note the debugger will be entered even the event of simple syntax errors (not preferable), the above is reversed through the following code:

```
> options(error = NULL)
```

ensuring consistency while debugging in R

In the case of random numbers, it is important to reproduce the same values in order to effectively debug while producing the same errors as the initial cause. The **set.seed( )** function controls the latter by reinitializing the random number sequence of a given value:

```
> runif(3)
[1] 0.1842674 0.7220917 0.4818511

> runif(3)
[1] 0.1563653 0.4570898 0.6093413

> runif(3)
[1] 0.55114431 0.05548648 0.67934008

> set.seed(8888)
> runif(3)
[1] 0.5775979 0.4588383 0.8354707

> set.seed(8888)
> runif(3)
```