Basics of Debugging R

Like any program, R will occasionally produce errors which are not easily understood. The classical method of debugging is to insert print statements at appropriate locations, which can be time consuming and inefficient.

One simple tool that is sometimes useful in R is the `traceback()` function. If a program fails, and it's not clear where or why, invoking `traceback()` may provide useful information.

However, it is often usefult to have a more interactive environment in which to examine and modify variables.

Interactive Debugging in R

R provides facilities for interactive debugging which can be invoked through a variety of methods. The basic function for debugging is `browser()`, which pauses the current execution, and provides an R interpreter, allowing you to view and modify variables, and then continue execution.

`browser()` can be invoked in several ways:

- insert a call to `browser()` in your R code.
- invoke the browser after each step of a function's execution using `debug()`
- invoke the browser if there's an error in the your program by calling `recover()` via `options(error=recover)`.
- temporarily modify a function to allow browsing (or other code of your choice) using `trace()`

Using `browser()`

When you are in the R browser, the following single letter commands have special meaning:

- `c` - continue: execute all of the current function
- `n` (or carriage return) - next: step through the current function a line at a time
- `Q` - quit: terminates debugging
- `control-C` return to top level - allows you to move among frames

You'll usually have to type "`n`" to get started once you're in the browser.

Once in the browser, any valid R commands can be entered. Typing `ls()` or `objects()` is especially useful. Note that to examine variables named `c`, `n` or `Q`, you need to use `print()` statements.

Inserting `browser()` into your function

If you place a call to `browser()` inside a function, execution will pause when the call is reached, allowing you to examine and modify variables. All valid R statements can be used.

Suppose we have the following function:

```
doit = function(x,y){
    z = x + y
    browser()
    return(sum(z > 10))
}
```

Now suppose we invoke the function:

```
> doit(1:8,5:12)
Called from: doit(1:8, 5:12)
Browse[1]> objects()
[1] "x" "y" "z"
Browse[1]> x
[1] 1 2 3 4 5 6 7 8
```

Using `debug()`

When you want to interact with your R program on a step-by-step basis, you can use the `debug()` function. `debug()` accepts a single argument, the name of a function, and the function is then flagged for debugging. To unflag a function, pass the function name to `undebug()`.

```
> debug(doit)
> doit(1:8,5:12)
debugging in: doit(1:8, 5:12)
debug: {
    z = x + y
    return(sum(z > 10))
}
Browse[1]>
```

The entire body of the function is printed when you first enter the function, and individual lines are printed as you step through using the "`n`" command.

Debugging Builtin Functions

The `debug()` command is not limited to user-written functions – you can pass any non-primitive function to `debug()`. As an example, suppose you are fitting a model with `glm`, and you would like to investigate the `glm.fit` function.

```
> debug(glm.fit)
> glm(y~x,data=mydata)
```

Once `glm.fit` is reached, the body of the function is printed, followed by the browser prompt.

```
Browse[1]> objects()
 [1] "control"   "etastart"  "family"    "intercept" "mustart"
 [6] "offset"    "start"     "weights"   "x"         "y"
Browse[1]> n
debug: x <- as.matrix(x)
Browse[1]> n
debug: xnames <- dimnames(x)[[2]]
Browse[1]>
```

Using `trace()`

The `trace()` function allows you to temporarily add arbitrary code to a function; `untrace()` allows you to remove that code. This allows inserting code in a function without permanently changing it.

The `trace()` function accepts the name of the function to be traced, a function or unevaluated evaluated expression to execute, and the line number at which to execute it. Line numbers inside a function are revealed by calling `list(body(function))`.

For complex tracing, including tracing inside of loops, the `edit=TRUE` argument can be passed to `trace()`. This will invoke a text editor, allowing you to insert tracing code wherever desired, and a call to `untrace()` will remove the code. This is especially convenient when dealing with built-in functions.

Profiling R code

Profiling a program means determining how much execution time a program spends in various different sections of code. In R, profiling is available on a per-function basis; *i.e.* you can only see which functions are being invoked, not which individual lines within those functions. But by designing your program in a clever way, you can often spot bottlenecks which might conveniently be converted to C code.

Profiling is turned on with the `Rprof` function; you can specify a filename to direct output to a particular file, or `NULL` to turn off profiling.
To see the results of the profiling, run:

```
R CMD Rprof filename
```

where `filename` is the name of the profiler's output (`Rprof.out` if you don't specify a filename.

Example of Profiling

Consider the problem of removing rows of a data frame if they contain any missing values. It is "common knowledge" that loops in R can be very slow. Consider three ways of removing rows with missing values:

```
fun1 = function(x){
   res = NULL
   n = nrow(x)
   for(i in 1:n)if(!any(is.na(x[i,])))
      res = rbind(res,x[i,])
   res
}
```

```
fun2 = function(x){
   n = nrow(x)
   res = matrix(0,n,ncol(x))
   k = 1
   for(i in 1:n)
          if(!any(is.na(x[i,]))){
             res[k,] = x[i,]
             k = k + 1
             }
   res[1:(k-1),]
}
```

```
fun3 = function(x){
    omit = F
    n = ncol(x)
    for(i in 1:n)
       omit = omit | is.na(x[,i])
    x[!omit,]
}
```


Example of Profiling R code

To facilitate timing and profiling, we can use the following script:

```
x = matrix(rnorm(20000000),100000,20)
x[x > 1.5] = NA

Rprof("method1.out")
print(unix.time(fun1(x)))
Rprof(NULL)

Rprof("method2.out")
print(unix.time(fun2(x)))
Rprof(NULL)

Rprof("method3.out")
print(unix.time(fun3(x)))
Rprof(NULL)
```

Results of Timing and Profiling

```
[1] 194.68  49.19 245.71   0.00   0.00
[1] 0.99 0.05 1.04 0.00 0.00
[1] 0.18 0.03 0.21 0.00 0.00
```

Timings indicate that the first method is indeed very slow. However, the other two loops were reasonably fast. To examine the profiles, we exit R and use a command like

```
R CMD Rprof filename
```

(The profiler can be called from within R through the `summaryRprof()` function, but it's more efficient to invoke it from the command line as shown above.)

Results for Method 1

```
Each sample represents 0.02 seconds.
Total run time: 232.280000000045 seconds.

Total seconds: time spent in function and callees.
Self seconds: time spent in function alone.

   %       total       %       self
 total    seconds     self    seconds    name
100.00    232.28      0.20      0.46     "fun1"
100.00    232.28      0.00      0.00     "eval.with.vis"
100.00    232.28      0.00      0.00     "source"
100.00    232.28      0.00      0.00     "print"
100.00    232.28      0.00      0.00     "unix.time"
100.00    232.28      0.00      0.00     "eval"
 99.22    230.46     99.21    230.44     "rbind"
  0.56      1.30      0.53      1.24     "any"
  0.03      0.06      0.03      0.06     "!"
  0.03      0.06      0.03      0.06     "is.na"
  0.01      0.02      0.01      0.02     "!="

   %       self        %       total
 self     seconds    total    seconds    name
 99.21    230.44     99.22    230.46     "rbind"
  0.53      1.24      0.56      1.30     "any"
  0.20      0.46    100.00    232.28     "fun1"
  0.03      0.06      0.03      0.06     "!"
  0.03      0.06      0.03      0.06     "is.na"
  0.01      0.02      0.01      0.02     "!="
```

Results for Method 2

```
Each sample represents 0.02 seconds.
Total run time: 0.980000000000001 seconds.

Total seconds: time spent in function and callees.
Self seconds: time spent in function alone.

   %       total       %       self
 total    seconds     self    seconds    name
100.00      0.98      0.00      0.00     "eval.with.vis"
100.00      0.98     42.86      0.42     "fun2"
100.00      0.98      0.00      0.00     "source"
100.00      0.98      0.00      0.00     "print"
100.00      0.98      0.00      0.00     "unix.time"
100.00      0.98      0.00      0.00     "eval"
 51.02      0.50     48.98      0.48     "any"
  4.08      0.04      4.08      0.04     "matrix"
  2.04      0.02      2.04      0.02     "!"
  2.04      0.02      2.04      0.02     "is.na"

   %       self        %       total
 self     seconds    total    seconds    name
 48.98      0.48     51.02      0.50     "any"
 42.86      0.42    100.00      0.98     "fun2"
  4.08      0.04      4.08      0.04     "matrix"
  2.04      0.02      2.04      0.02     "!"
  2.04      0.02      2.04      0.02     "is.na"
```


Results for Method 3

```
Each sample represents 0.02 seconds.
Total run time: 0.18 seconds.

Total seconds: time spent in function and callees.
Self seconds: time spent in function alone.

   %       total       %       self
 total    seconds     self    seconds    name
100.00      0.18     33.33      0.06     "fun3"
100.00      0.18      0.00      0.00     "eval.with.vis"
100.00      0.18      0.00      0.00     "eval"
100.00      0.18      0.00      0.00     "unix.time"
100.00      0.18      0.00      0.00     "print"
100.00      0.18      0.00      0.00     "source"
 44.44      0.08     44.44      0.08     "|"
 22.22      0.04     22.22      0.04     "is.na"

   %       self        %       total
 self     seconds    total    seconds    name
 44.44      0.08     44.44      0.08     "|"
 33.33      0.06    100.00      0.18     "fun3"
 22.22      0.04     22.22      0.04     "is.na"
```

Conclusions from Profiling Example

Usually the second part of the profiling output (time spent in function alone) is most useful.

Since the difference between method 1 and method 2 was the replacing the repeated `rbind()` call with a single call to `matrix()` we can compare the execution time for `rbind` in method 1 (230 seconds) with the execution time for `matrix` in method 2 (.04 seconds). This speedup is due to preallocating the result matrix.

Looking at method 3's profile, it appears that it improves on method 2 primarily by eliminating the need to repeatedly call `any()` inside the loop, along with the added speed of the "`|`" operator compared to the logic used in method 2. Time spent in primitive (i.e. non-function) execution is lumped together in the line corresponding to the function's name: 0.46 seconds for method 1, 0.42 seconds for method 2, and .06 seconds for method 3.