R: Debugging

140.776 Statistical Computing

September 8, 2011

Debugging

A grammatically correct program may give you incorrect results due to logic errors. In case such errors (i.e. bugs) occur, you need to find out why and where they occur so that you can fix them. The procedure to identify and fix bugs is called "debugging".

R provides a number of tools for debugging:

- traceback()
- debug()
- browser()
- trace()
- recover()

traceback()

- When an R function fails, an error is printed to the screen.
- Immediately after the error, you can call traceback() to see in which function the error occurred.
- The traceback() function prints the list of functions that were called before the error occurred.
- The functions are printed in reverse order.

traceback()

Example:

```
f<-function(x) {
    r<-x-g(x)
    r
}

g<-function(y) {
  r<-y*h(y)
  r
}

h<-function(z) {
    r<-log(z)
    if(r<10)
        r^2
    else
        r^3
}
```

traceback()

```
> f(-1)
Error in if (r < 10) r^2 else r^3 : missing value where TRUE/FALSE needed
In addition: Warning message:
In log(z) : NaNs produced

> traceback()
3: h(y)
2: g(x)
1: f(-1)
```

debug()

traceback() does not tell you where in the function the error occurred. In order to know which line causes the error, you may want to step through the function using debug().

- debug(foo) flags the function foo() for debugging.
- undebug(foo) unflags the function.
- When a function is flagged for debugging, each statement in the function is executed one at a time. After a statement is executed, the function suspends and you can interact with the environment.
- After you obtained necessary information from the enviroment, you can let the function to execute the next statement. In this way, you can check the function line-by-line.

debug()

Example: compute sum of squared error $SS = \sum_{i=1}^{n}(x_i - \mu)^2$.

```
## compute sum of squares
SS<-function(mu,x) {
  d<-x-mu
  d2<-d^2
  ss<-sum(d2)
  ss
}
```

debug()

```
## set seed to get reproducible results
> set.seed(100)
> x<-rnorm(100)
> SS(1,x)
[1] 202.5615

## now start debugging
> debug(SS)
> SS(1,x)
debugging in: SS(1, x)
debug: {
    d <- x - mu
    d2 <- d^2
    ss <- sum(d2)
    ss
}
attr(,"srcfile")
C:\Users\jihk\doc\courses\StatisticalComputing\lecture5\debugex2.R
Browse[1]>
```

debug()

After you see the "Browse[1]>" prompt, you can do different things:

- Typing n executes the current line and prints the next one;
- Typing c executes the rest of the function without stopping;
- Typing Q quits the debugging;
- Typing where tells where you are in the function call stack;
- Typing ls() list all objects in the local environment;
- Typing an object name or print(<object name>) tells you current value of the object. If your object has name n, c or Q, you have to use print() to see their values.

debug(): typing n

```
> SS(1,x)
debugging in: SS(1, x)
debug: {
    d <- x - mu
    d2 <- d^2
    ss <- sum(d2)
    ss
}
attr(,"srcfile")
C:\Users\jihk\doc\courses\StatisticalComputing\lecture5\debugex2.R
Browse[1]> n
debug: d <- x - mu
Browse[1]> n
debug: d2 <- d^2
Browse[1]> n
debug: ss <- sum(d2)
Browse[1]> n
debug: ss
Browse[1]> n
exiting from: SS(1, x)
[1] 202.5615
>
```

debug(): typing c

```
> SS(1,x)
debugging in: SS(1, x)
debug: {
    d <- x - mu
    d2 <- d^2
    ss <- sum(d2)
    ss
}
attr(,"srcfile")
C:\Users\jihk\doc\courses\StatisticalComputing\lecture5\debugex2.R
Browse[1]> n
debug: d <- x - mu
Browse[1]> c
exiting from: SS(1, x)
[1] 202.5615
>
```

debug(): ls, print, where, Q

```
> SS(1,x)
debugging in: SS(1, x)
debug: {
    d <- x - mu
    d2 <- d^2
...
Browse[1]> n
debug: d <- x - mu  ## the next command
Browse[1]> ls()     ## current environment
[1] "mu" "x"       ## there is no d
Browse[1]> n        ## go one step
debug: d2 <- d^2    ## the next command
Browse[1]> ls()     ## current environment
[1] "d"  "mu" "x"   ## d has been created
Browse[1]> d[1:3]   ## first three elements of d
[1] -1.5021924 -0.8684688 -1.0789171
Browse[1]> hist(d)  ## histogram of d
Browse[1]> where    ## current position in call stack
where 1: SS(1, x)
Browse[1]> n
debug: ss <- sum(d2)
Browse[1]> Q        ## quit
>
```

debug(): ls, print, where, Q

```
> SS(1,x)
debugging in: SS(1, x)
debug: {
    d <- x - mu
    d2 <- d^2
...
Browse[1]> n
debug: ss <- sum(d2)
Browse[1]> ls()
[1] "d"  "d2" "mu" "x"
Browse[1]> print(d2[1:2])   ## print an object
[1] 2.2565819 0.7542381
Browse[1]> y<-d^3           ## create a new object
Browse[1]> ls()
[1] "d"  "d2" "mu" "x"  "y"
Browse[1]> c
exiting from: SS(1, x)
[1] 202.5615
> undebug(SS)               ## remove debug label
> SS(1,x)                   ## now call SS again will
[1] 202.5615                ## not start the debugger
```

debug()

You can label a function for debugging while debugging another function

```
f<-function(x) {
    r<-x-g(x)
  r
}

g<-function(y) {
  r<-y*h(y)
  r
}

h<-function(z) {
  r<-log(z)
  if(r<10)
    r^2
  else
    r^3
}
```

debug()

You can label a function for debugging while debugging another function

```
## If you only debug f, you will not go into g
  > debug(f)
  > f(-1)
  debugging in: f(-1)
  debug: {
      r <- x - g(x)
      r
  }
  attr(,"srcfile")
  C:\Users\jihk\doc\courses\StatisticalComputing\lecture5\debugex1.R
  Browse[1]> n
  debug: r <- x - g(x)
  Browse[1]> n
  Error in if (r < 10) r^2 else r^3 : missing value where TRUE/FALSE needed
  In addition: Warning message:
  In log(z) : NaNs produced
  >
```

debug()

```
## You can label g and h for debugging when you debug f
  > f(-1)
  debugging in: f(-1)
  ...
  Browse[1]> n
  debug: r <- x - g(x)
  Browse[1]> debug(g)
  Browse[1]> n
  debugging in: g(x)
  debug: {
      r <- y * h(y)
      r
  }
  attr(,"srcfile")
  C:\Users\jihk\doc\courses\StatisticalComputing\lecture5\debugex1.R
  Browse[1]> n
  debug: r <- y * h(y)
  Browse[1]> debug(h)
  Browse[1]> n
  debugging in: h(y)
  ...
```

debug()

```
debugging in: h(y)
  ...
  Browse[1]> n
  debug: r <- log(z)
  Browse[1]> n
  debug: if (r < 10) r^2 else r^3
  Browse[1]> n
  Error in if (r < 10) r^2 else r^3 : missing value where TRUE/FALSE needed
  >
```

browser()

- Inserting a call to browser() in a function will pause the execution of a function at the point where browser() is called.
- Similar to using debug() except you can control where execution gets paused.

browser()

```
h<-function(z) {
    browser() ## a break point inserted here
    r<-log(z)
    if(r<10)
        r^2
    else
        r^3
}
```

browser()

```
> f(-1)
Called from: h(y)
Browse[1]> ls()
[1] "z"
Browse[1]> z
[1] -1
Browse[1]> n
debug: r <- log(z)
Browse[1]> n
debug: if (r < 10) r^2 else r^3
Browse[1]> ls()
[1] "r" "z"
Warning message:
In log(z) : NaNs produced
Browse[1]> r
[1] NaN
Browse[1]> c
Error in if (r < 10) r^2 else r^3 : missing value where TRUE/FALSE needed
>
```

trace()

- Calling trace() on a function allows the user to insert bits of code into a function.
- The syntax for trace() is a bit strange for first time users.
- You might be better off using debug()

trace()

```
> as.list(body(h))
[[1]]
`{`

[[2]]
r <- log(z)

[[3]]
if (r < 10) r^2 else r^3

attr(,"srcfile")
C:\Users\jihk\doc\courses\StatisticalComputing\lecture5\debugex1.R
```

trace()

```
> trace("h",quote(if(is.nan(r)) {browser()}), at=3, print=FALSE)
> f(1)
[1] 1
> f(-1)
Called from: eval(expr, envir, enclos)
Browse[1]> ls()
[1] "r" "z"
Warning message:
In log(z) : NaNs produced
Browse[1]> r
[1] NaN
Browse[1]> z
[1] -1
Browse[1]> c
...
>
```

trace()

```
> trace("h",quote(if(z<0) {z<-1}), at=2, print=FALSE)
[1] "h"
> f(-1)
[1] -1
```

untrace()

```
> trace("h",quote(if(is.nan(r)) {browser()}), at=3, print=FALSE)
[1] "h"
> f(-1)
Called from: eval(expr, envir, enclos)
Browse[1]> c
Error in if (r < 10) r^2 else r^3 : missing value where TRUE/FALSE needed
In addition: Warning message:
In log(z) : NaNs produced

> untrace(h)
> f(-1)
Error in if (r < 10) r^2 else r^3 : missing value where TRUE/FALSE needed
In addition: Warning message:
In log(z) : NaNs produced
```

recover()

When you are debugging a function, recover() allows you to check variables in upper level functions.

recover()

```
> trace("h",quote(if(is.nan(r)) {recover()}), at=3, print=FALSE)
[1] "h"
> f(-1)

Enter a frame number, or 0 to exit

1: f(-1)
2: g(x)
3: h(y)

Selection: 1
Called from: eval(expr, envir, enclos)
Browse[1]> ls()
[1] "x"
Warning message:
In log(z) : NaNs produced
Browse[1]> x
[1] -1
Browse[1]> c
```

recover()

```
Enter a frame number, or 0 to exit

1: f(-1)
2: g(x)
3: h(y)

Selection: 2
Called from: eval(expr, envir, enclos)
Browse[1]> ls()
[1] "y"
Browse[1]> y
[1] -1
Browse[1]> c
```

recover()

```
Enter a frame number, or 0 to exit

1: f(-1)
2: g(x)
3: h(y)

Selection: 3
Called from: eval(expr, envir, enclos)
Browse[1]> ls()
[1] "r" "z"
Browse[1]> r
[1] NaN
Browse[1]> z
[1] -1
Browse[1]> c
```

recover()

```
Enter a frame number, or 0 to exit

1: f(-1)
2: g(x)
3: h(y)

Selection: 0
Error in if (r < 10) r^2 else r^3 : missing value where TRUE/FALSE needed
```

recover()

- recover() can be used as an error handler, set using options() (e.g. options(error=recover)).
- When a function throws an error, execution is halted at the point of failure. You can browse the function calls and examine the environment to find the source of the problem.

General guidelines

- Don't underestimate the usefulness of print() and cat().
- Try not to develop an unhealthy relationship with the debugger.
- Think first, then program.

Sorting: application of recursive functions

Sorting numbers in a vector. Suppose you have a vector:

```
38 27 43 3 9 82 10
```

A simple way to sort it is by insertion sort:

```
27 38 43  3  9 82 10 ## sort 27
27 38 43  3  9 82 10 ## sort 43
 3 27 38 43  9 82 10 ## sort  3
 3  9 27 38 43 82 10 ## sort  9
 3  9 27 38 43 82 10 ## sort 82
 3  9 10 27 38 43 82 ## sort 10
```

In the worst case, it requires $O(n^2)$ operations (i.e. the worst-case time complexity of insertion sort is $O(n^2)$).

Sorting: merge sort

Merge sort is a better algorithm whose worst-case time complexity is $O(nlog(n))$. It uses a divide-and-conquer strategy:

1. If the vector has only one element, done; otherwise
2. divide the vector into two subvectors, each containing about half of the elements;
3. sort each subvector by merge sort;
4. merge the two subvectors into a sorted vector.

Sorting: merge sort

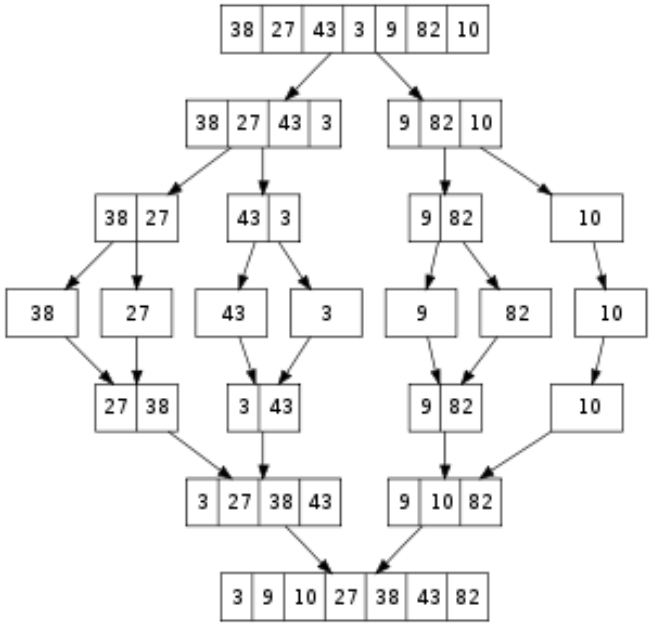

Figure source: http://en.wikipedia.org/wiki/Mergesort

Sorting: merge sort

Merge sort require at most n comparisons at each dividing step. The input vector will be divided $log_2(n)$ times. Therefore, the total number of operations in the worst case is proportional to $n * log_2(n)$.

Merge sort is a recursive function, the pseudocode for implementing the function is:

```
mergesort<-function(x) {
        xlen<-length(x)
        if(xlen == 1) {
                ## no need to sort
                result<-x
        } else {
                mid<-as.integer(xlen/2)
                ## sort left and right subvectors
                xleft<-mergesort(x[1:mid])
                xright<-mergesort(x[(mid+1):xlen])
                ## merge into one vector
                result<-merge(xleft,xright)
        }
}
```