Debugging Functions in R Statistics 506

Debugging

When writing simple functions in R, it is often sufficient to write the code external to a function, then once it is working, wrap it in a function call.

However, as your functions get more advanced, it becomes more common to need to figure out bugs in the code, and extracting the code from the function body each time is inefficient.

Here are a few ways you can debug R functions, starting with the simplest and ending with the most formal.

`print` statements

The simplest way to start debugging a function is to use `print` to check the status of objects to ensure things are proceeding as you expect.

```
#' Calculate sample skewness
#' @param x a numeric vector
#' @return skewness
skewness <- function(x) {
  xbar <- mean(x)
  print(paste("xbar:", xbar))
  xc <- x - xbar
  print(paste("xc:", paste(xc, collapse = ", ")))
  num <- sum(xc^3)
  denom <- sum(xc^2)
  print(paste("Num:", num))
  print(paste("Denom:", denom))
  return(num/(denom^(3/2)))
}
skewness(sample(1:100, 10, TRUE))
```

```
[1] "xbar: 35.9"
[1] "xc: -2.9, -34.9, -0.899999999999999, 5.1, 2.1, 44.1, -18.9, 26.1, -26.9, 7.1"
[1] "Num: 35295.48"
[1] "Denom: 5014.9"
```

```
[1] 0.09938611
```

Note the double `paste` to first collapse the entries in the `xc` vector into a single string, before adding a prefix.

Saving objects into global environment

We discussed earlier avoiding using the `<<-` assignment. Debugging is one case where it is permissible - just be sure to remove it once you're done debugging! The `<<-` assigns an object into the global environment, regardless of being inside a function. (Recall that `<-` inside a function assigns the object in the environment of the function, and that environment (and all its objects) get deleted when the function exits, via error or `return`).

```
#' Calculate sample skewness
#' @param x a numeric vector
#' @return skewness
skewness <- function(x) {
  xbar <- mean(x)
  xc <- x - xbar
  num <- sum(xc^3)
  denom <- sum(xc^2)
  num <<- num
  denom <<- denom
  return(num/(denom^(3/2)))
}
num
```

```
Error in eval(expr, envir, enclos): object 'num' not found
```

```
s <- skewness(sample(1:100, 10, TRUE))
c(num, denom)
```

```
[1] 21271.32 10178.90
```

Be sure to remove these objects once you are done! If not, you can miss obvious issues.

```
broken_function <- function(y) {
  num < -3
  (y - mean(y))/num
}
broken_function(1:5)
```

```
[1] -9.402331e-05 -4.701166e-05  0.000000e+00  4.701166e-05  9.402331e-05
```

This function should error! But it doens't, since `num` still exists from earlier.

```
rm(num, denom)
broken_function(1:5)
```

```
Error in broken_function(1:5): object 'num' not found
```

traceback

Whenever you call a function in R, you enter the **call stack**. Consider the following:

```
foo <- function(x) {
  x <- x + 1
  bar(x)
}

bar <- function(x) {
  a <- baz(x)
  return(a)
}

baz <- function(x) {
  sum(x, "a")
}
```

What happens if we call `foo(3)`? First, we enter the function `foo`. Inside here we call `bar`, so now we're inside `bar` inside `foo`. Then inside `bar` we call `baz`, so we're insde `baz` inside `bar` inside `foo`. The stack thus consists of 1: `foo`, 2: `bar`, 3: `baz`.

Knowing where we are in this callstack can help a lot with debugging functions:

```
foo(3)
```

```
Error in sum(x, "a"): invalid 'type' (character) of argument
```

```
traceback()
```

The numbers tell you which line of the calling function each sub-function is referenced at: the "#3" means the call to `bar` occurs on the 3rd line of `foo` (including the definition line).

This can help you to identify what specific function is erroring (this is especially useful during modeling steps as there can be a very large call stack), or what other functions in the callstack may be causing the issue.

The "browser"

Formal debugging, stepping through a function line-by-line as if it were purely interactive, can be performed by using the "browser". There are two ways to enter the browser:

1. Use the `debug` function to flag your function. E.g. `debug(foo)`. The next time `foo()` is called, the browser will be started. You can `undebug(foo)` to turn off the browser for it, or redefine the function.
2. Especially if you have a long function, you can choose to enter the browser at any line in the function by including a call to `browser()`. For example,

   ```
   foo <- function(x) {
     x <- abs(x)
     browser()
     x <- x + 1
     return(x)
   }
   ```

   Calling `foo()` will enter the browser at the line `x <- x + 1`.

Using the browser

Here's the what the browser looks like:

```
foo <- function(x, y = 3) {
  browser()
  q <- 3
```

```
  z <- bar(x, y)
  return(q + z)
}
bar <- function(a, b) {
  out <- b
  for (i in 1:a) {
    out <- out + b
  }
  out
}
```

```
> foo(2)
Called from: foo(2)
Browse[1]> debug at #3: bar(x, y)
Browse[2]>
```

We see the text `Browse[n]` prepended to the typical `>` prompt. The number there represents the position in the call stack, here we called `foo` directly from the global environment (`1`), and then `foo`'s environment is on top of it (`2`).

All normal R commands:

```
Browse[2]> 2 + 2
[1] 4
Browse[2]> ls()
[1] "x" "y"
```

Note that we are in an environment for `foo`, *not* the global environment - the only objects that exist are those that are passed into `foo`. You can examine these objects to see their current status at any time. As you create new objects, they'll exist in this environment as well.

In addition to normal R commands, you can use the following special command:

- `Q`: Terminate the browser, not running any further code.
- `c`: Terminate the browser by running the remaining code to completion.
- `n`: Run the next line of code
- `s`: Run the next line of code, but step into any function calls.
- Blank: Run `s` or `n`, whichever was most recent.
- `where`: Print the call stack.

An example. First we `n` to the call to `bar`, the `s`tep into that call (note the change to the prompt of `Browse[3]`), go through a few steps of that, then use `c` to finish running the code.

```
> foo(2)
Called from: foo(2)
Browse[1]> debug at #3: q <- 3
Browse[2]> n
debug at #4: z <- bar(x, y)
Browse[2]> s
debugging in: bar(x, y)
debug at #1: {
    out <- b
    for (i in 1:a) {
        out <- out + b
    }
    out
}
Browse[3]> n
debug at #2: out <- b
Browse[3]> n
debug at #3: for (i in 1:a) {
    out <- out + b
}
Browse[3]> c
exiting from: bar(x, y)
debug at #5: return(q + z)
Browse[2]> c
[1] 12
```