R Programming for Data Science

Roger D. Peng

20. Debugging

20.1 Something's Wrong!

Watch a video of this section[1] (note that this video differs slightly from the material presented here)

R has a number of ways to indicate to you that something's not right. There are different levels of indication that can be used, ranging from mere notification to fatal error. Executing any function in R may result in the following *conditions*.

- `message`: A generic notification/diagnostic message produced by the `message()` function; execution of the function continues
- `warning`: An indication that something is wrong but not necessarily fatal; execution of the function continues. Warnings are generated by the `warning()` function
- `error`: An indication that a fatal problem has occurred and execution of the function stops. Errors are produced by the `stop()` function.
- `condition`: A generic concept for indicating that something unexpected has occurred; programmers can create their own custom conditions if they want.

Here is an example of a warning that you might receive in the course of using R.

```
> log(-1)
Warning in log(-1): NaNs produced
[1] NaN
```

This warning lets you know that taking the log of a negative number results in a `NaN` value because you can't take the log of negative numbers. Nevertheless, R doesn't give an error, because it has a useful value that it can return, the `NaN` value. The warning is just there to let you know that something unexpected happen. Depending on what you are programming, you may have intentionally taken the log of a negative number in order to move on to another section of code.

Here is another function that is designed to print a message to the console depending on the nature of its input.

[1] https://youtu.be/LHQxbRInyyc

```
> printmessage <- function(x) {
+         if(x > 0)
+                 print("x is greater than zero")
+         else
+                 print("x is less than or equal to zero")
+         invisible(x)
+ }
```

This function is simple—it prints a message telling you whether `x` is greater than zero or less than or equal to zero. It also returns its input *invisibly*, which is a common practice with "print" functions. Returning an object invisibly means that the return value does not get auto-printed when the function is called.

Take a hard look at the function above and see if you can identify any bugs or problems.

We can execute the function as follows.

```
> printmessage(1)
[1] "x is greater than zero"
```

The function seems to work fine at this point. No errors, warnings, or messages.

```
> printmessage(NA)
Error in if (x > 0) print("x is greater than zero") else print("x is less than or equal to\
 zero"): missing value where TRUE/FALSE needed
```

What happened?

Well, the first thing the function does is test if `x > 0`. But you can't do that test if `x` is a `NA` or `NaN` value. R doesn't know what to do in this case so it stops with a fatal error.

We can fix this problem by anticipating the possibility of `NA` values and checking to see if the input is `NA` with the `is.na()` function.

```
> printmessage2 <- function(x) {
+         if(is.na(x))
+                 print("x is a missing value!")
+         else if(x > 0)
+                 print("x is greater than zero")
+         else
+                 print("x is less than or equal to zero")
+         invisible(x)
+ }
```

Now we can run the following.

```
> printmessage2(NA)
[1] "x is a missing value!"
```

And all is fine.

Now what about the following situation.

```
> x <- log(c(-1, 2))
Warning in log(c(-1, 2)): NaNs produced
> printmessage2(x)
Warning in if (is.na(x)) print("x is a missing value!") else if (x > 0)
print("x is greater than zero") else print("x is less than or equal to
zero"): the condition has length > 1 and only the first element will be
used
[1] "x is a missing value!"
```

Now what?? Why are we getting this warning? The warning says "the condition has length > 1 and only the first element will be used".

The problem here is that I passed `printmessage2()` a vector `x` that was of length 2 rather then length 1. Inside the body of `printmessage2()` the expression `is.na(x)` returns a vector that is tested in the `if` statement. However, `if` cannot take vector arguments so you get a warning. The fundamental problem here is that `printmessage2()` is not *vectorized*.

We can solve this problem two ways. One is by simply not allowing vector arguments. The other way is to vectorize the `printmessage2()` function to allow it to take vector arguments.

For the first way, we simply need to check the length of the input.

```
> printmessage3 <- function(x) {
+         if(length(x) > 1L)
+                 stop("'x' has length > 1")
+         if(is.na(x))
+                 print("x is a missing value!")
+         else if(x > 0)
+                 print("x is greater than zero")
+         else
+                 print("x is less than or equal to zero")
+         invisible(x)
+ }
```

Now when we pass `printmessage3()` a vector we should get an error.

```
> printmessage3(1:2)
Error in printmessage3(1:2): 'x' has length > 1
```

Vectorizing the function can be accomplished easily with the `Vectorize()` function.

```
> printmessage4 <- Vectorize(printmessage2)
> out <- printmessage4(c(-1, 2))
[1] "x is less than or equal to zero"
[1] "x is greater than zero"
```

You can see now that the correct messages are printed without any warning or error. Note that I stored the return value of `printmessage4()` in a separate R object called `out`. This is because when I use the `Vectorize()` function it no longer preserves the invisibility of the return value.

20.2 Figuring Out What's Wrong

The primary task of debugging any R code is correctly diagnosing what the problem is. When diagnosing a problem with your code (or somebody else's), it's important first understand what you were expecting to occur. Then you need to idenfity what *did* occur and how did it deviate from your expectations. Some basic questions you need to ask are

- What was your input? How did you call the function?
- What were you expecting? Output, messages, other results?
- What did you get?
- How does what you get differ from what you were expecting?
- Were your expectations correct in the first place?
- Can you reproduce the problem (exactly)?

Being able to answer these questions is important not just for your own sake, but in situations where you may need to ask someone else for help with debugging the problem. Seasoned programmers will be asking you these exact questions.

20.3 Debugging Tools in R

Watch a video of this section[2]

R provides a number of tools to help you with debugging your code. The primary tools for debugging functions in R are

[2] https://youtu.be/h9rs6-Cwwto

- `traceback()`: prints out the function call stack after an error occurs; does nothing if there's no error
- `debug()`: flags a function for "debug" mode which allows you to step through execution of a function one line at a time
- `browser()`: suspends the execution of a function wherever it is called and puts the function in debug mode
- `trace()`: allows you to insert debugging code into a function a specific places
- `recover()`: allows you to modify the error behavior so that you can browse the function call stack

These functions are interactive tools specifically designed to allow you to pick through a function. There's also the more blunt technique of inserting `print()` or `cat()` statements in the function.

20.4 Using `traceback()`

Watch a video of this section[3]

The `traceback()` function prints out the *function call stack* after an error has occurred. The function call stack is the sequence of functions that was called before the error occurred.

For example, you may have a function `a()` which subsequently calls function `b()` which calls `c()` and then `d()`. If an error occurs, it may not be immediately clear in which function the error occurred. The `traceback()` function shows you how many levels deep you were when the error occurred.

```
> mean(x)
Error in mean(x) : object 'x' not found
> traceback()
1: mean(x)
```

Here, it's clear that the error occurred inside the `mean()` function because the object `x` does not exist.

The `traceback()` function must be called immediately after an error occurs. Once another function is called, you lose the traceback.

Here is a slightly more complicated example using the `lm()` function for linear modeling.

[3] https://youtu.be/VT9ZxCp6o-I

```
> lm(y ~ x)
Error in eval(expr, envir, enclos) : object 'y' not found
> traceback()
7: eval(expr, envir, enclos)
6: eval(predvars, data, env)
5: model.frame.default(formula = y ~ x, drop.unused.levels = TRUE)
4: model.frame(formula = y ~ x, drop.unused.levels = TRUE)
3: eval(expr, envir, enclos)
2: eval(mf, parent.frame())
1: lm(y ~ x)
```

You can see now that the error did not get thrown until the 7th level of the function call stack, in which case the `eval()` function tried to evaluate the formula `y ~ x` and realized the object `y` did not exist.

Looking at the traceback is useful for figuring out roughly where an error occurred but it's not useful for more detailed debugging. For that you might turn to the `debug()` function.

20.5 Using `debug()`

The `debug()` function initiates an interactive debugger (also known as the "browser" in R) for a function. With the debugger, you can step through an R function one expression at a time to pinpoint exactly where an error occurs.

The `debug()` function takes a function as its first argument. Here is an example of debugging the `lm()` function.

```
> debug(lm)      ## Flag the 'lm()' function for interactive debugging
> lm(y ~ x)
debugging in: lm(y ~ x)
debug: {
    ret.x <- x
    ret.y <- y
    cl <- match.call()
    ...
    if (!qr)
        z$qr <- NULL
    z
}
Browse[2]>
```

Now, every time you call the `lm()` function it will launch the interactive debugger. To turn this behavior off you need to call the `undebug()` function.

The debugger calls the browser at the very top level of the function body. From there you can step through each expression in the body. There are a few special commands you can call in the browser:

- `n` executes the current expression and moves to the next expression
- `c` continues execution of the function and does not stop until either an error or the function exits
- `Q` quits the browser

Here's an example of a browser session with the `lm()` function.

```
Browse[2]> n   ## Evalute this expression and move to the next one
debug: ret.x <- x
Browse[2]> n
debug: ret.y <- y
Browse[2]> n
debug: cl <- match.call()
Browse[2]> n
debug: mf <- match.call(expand.dots = FALSE)
Browse[2]> n
debug: m <- match(c("formula", "data", "subset", "weights", "na.action",
    "offset"), names(mf), 0L)
```

While you are in the browser you can execute any other R function that might be available to you in a regular session. In particular, you can use `ls()` to see what is in your current environment (the function environment) and `print()` to print out the values of R objects in the function environment.

You can turn off interactive debugging with the `undebug()` function.

```
undebug(lm)    ## Unflag the 'lm()' function for debugging
```

20.6 Using `recover()`

The `recover()` function can be used to modify the error behavior of R when an error occurs. Normally, when an error occurs in a function, R will print out an error message, exit out of the function, and return you to your workspace to await further commands.

With `recover()` you can tell R that when an error occurs, it should halt execution at the exact point at which the error occurred. That can give you the opportunity to poke around in the environment in which the error occurred. This can be useful to see if there are any R objects or data that have been corrupted or mistakenly modified.

```
> options(error = recover)    ## Change default R error behavior
> read.csv("nosuchfile")      ## This code doesn't work
Error in file(file, "rt") : cannot open the connection
In addition: Warning message:
In file(file, "rt") :
  cannot open file 'nosuchfile': No such file or directory

Enter a frame number, or 0 to exit

1: read.csv("nosuchfile")
2: read.table(file = file, header = header, sep = sep, quote = quote, dec =
3: file(file, "rt")

Selection:
```

The `recover()` function will first print out the function call stack when an error occurrs. Then, you can choose to jump around the call stack and investigate the problem. When you choose a frame number, you will be put in the browser (just like the interactive debugger triggered with `debug()`) and will have the ability to poke around.

20.7 Summary

- There are three main indications of a problem/condition: `message`, `warning`, `error`; only an `error` is fatal
- When analyzing a function with a problem, make sure you can reproduce the problem, clearly state your expectations and how the output differs from your expectation
- Interactive debugging tools `traceback`, `debug`, `browser`, `trace`, and `recover` can be used to find problematic code in functions
- Debugging tools are not a substitute for thinking!