**Introduction**

As pointed out on the R hub’s website, reading the actual code, not just the documentation is a great way to learn more about programming and implementation details. But there is one more activity to get even more hands-on experience and understanding of the code in practice.

In this post, we provide tips on how to interactively debug R code step-by-step and investigate the values of objects in the middle of function execution. We will look at doing this for both exported and non-exported functions from different packages. We will also look at interactively debugging generics and methods, using functionality provided by base R.

**Interactively examining functions with debug() and debugonce()**

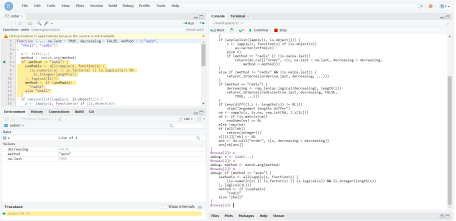
The 2 key functions we will be using for our interactive investigation of code are debug() and debugonce(). When debug() is called on a function, it will set a debugging flag on that function. When the function is executed, the execution will proceed one step at a time, giving us the option to investigate exactly what is going on in the context of that function call similarly to placing browser() at a certain point in our code.

Let us see a quick example:

debug(order)

order(10:1)

When running the second line, the code execution will stop inside order() and we can freely run the function line by line.



Debugging an R function interactively with debugonce()

When we no longer want to have the function flagged for debugging, call undebug():

undebug(order)

Alternatively, if we only want to have the function in debug mode for one execution, we can call debugonce() on the function. This approach may also be safer due to no need to undebug() later:

debugonce(order)

order(10:1)

**Debugging non-exported functions using :::**

The great thing about debug() and debugonce() is that they allow us to interactively investigate not just the code that we are currently writing, but any interpreted R function. To debug functions not even exported from package namespaces, we can use :::. For example, we normally cannot access the list\_rmds() function from the blogdown package as it is not exported.

# This will not work

library(blogdown)

debugonce(list\_rmds)

## Error in debugonce(list\_rmds): object 'list\_rmds' not found

# This will not work either

debugonce(blogdown::list\_rmds)

## Error: 'list\_rmds' is not an exported object from 'namespace:blogdown'

If we need to, we can still debug it using ::: to access it in the package namespace:

# This will work

debugonce(blogdown:::list\_rmds)

This is particularly useful when debugging nested calls inside package code, which tend to use unexported functions.

**Conveniently debugging methods with debugcall()**

Many R functions are implemented as S3 generics, that will call the proper method based on the signature of the arguments. A good example of this approach is aggregate(). Looking at its code, we see it only dispatches to the proper method based on the arguments provided:

body(stats::aggregate)

## UseMethod("aggregate")

Using debug(aggregate) would therefore not be very useful for interactive investigation, as we most likely want to look at the method that is called to actually see what is going on.

For this purpose, we can use debugcall(), which will conveniently take us directly to the method. In the following case, it is the data.frame method of the aggregate() generic:

eval(debugcall(

aggregate(mtcars["hp"], mtcars["carb"], FUN = mean),

once = TRUE

))

As seen above, we can also use the once = TRUE argument to only debug the call once.

For more technical details, the reference provided by ?debugcall is a great resource. This is also true for ?debug and ?trace which I also strongly recommend reading.

**Inserting debugging code anywhere inside a function body with trace()**

If debugonce() and friends are not sufficient for our purposes and we want to insert advanced debugging code at different places within a function body, we can use trace() to do just that.

Imagine for example we would like to investigate a specific place in the code of the aforementioned stats::aggregate.data.frame method. First, we can explore the function body:

as.list(body(stats::aggregate.data.frame))

## [[1]]

## `{`

##

## [[2]]

## if (!is.data.frame(x)) x <- as.data.frame(x)

##

## [[3]]

## FUN <- match.fun(FUN)

##

## [[4]]

## if (NROW(x) == 0L) stop("no rows to aggregate")

##

## [[5]]

## if (NCOL(x) == 0L) {

## x <- data.frame(x = rep(1, NROW(x)))

## return(aggregate.data.frame(x, by, function(x) 0L)[seq\_along(by)])

## }

##

## [[6]]

## if (!is.list(by)) stop("'by' must be a list")

##

## [[7]]

## if (is.null(names(by)) && length(by)) names(by) <- paste0("Group.",

## seq\_along(by)) else {

## nam <- names(by)

## ind <- which(!nzchar(nam))

## names(by)[ind] <- paste0("Group.", ind)

## }

##

## [[8]]

## if (any(lengths(by) != NROW(x))) stop("arguments must have same length")

##

## [[9]]

## y <- as.data.frame(by, stringsAsFactors = FALSE)

##

## [[10]]

## keep <- complete.cases(by)

##

## [[11]]

## y <- y[keep, , drop = FALSE]

##

## [[12]]

## x <- x[keep, , drop = FALSE]

##

## [[13]]

## nrx <- NROW(x)

##

## [[14]]

## ident <- function(x) {

## y <- as.factor(x)

## l <- length(levels(y))

## s <- as.character(seq\_len(l))

## n <- nchar(s)

## levels(y) <- paste0(strrep("0", n[l] - n), s)

## as.character(y)

## }

##

## [[15]]

## grp <- lapply(y, ident)

##

## [[16]]

## multi.y <- !drop && ncol(y)

##

## [[17]]

## if (multi.y) {

## lev <- lapply(grp, function(e) sort(unique(e)))

## y <- as.list(y)

## for (i in seq\_along(y)) y[[i]] <- y[[i]][match(lev[[i]],

## grp[[i]])]

## eGrid <- function(L) expand.grid(L, KEEP.OUT.ATTRS = FALSE,

## stringsAsFactors = FALSE)

## y <- eGrid(y)

## }

##

## [[18]]

## grp <- if (ncol(y)) {

## names(grp) <- NULL

## do.call(paste, c(rev(grp), list(sep = ".")))

## } else integer(nrx)

##

## [[19]]

## if (multi.y) {

## lev <- as.list(eGrid(lev))

## names(lev) <- NULL

## lev <- do.call(paste, c(rev(lev), list(sep = ".")))

## grp <- factor(grp, levels = lev)

## } else y <- y[match(sort(unique(grp)), grp, 0L), , drop = FALSE]

##

## [[20]]

## nry <- NROW(y)

##

## [[21]]

## z <- lapply(x, function(e) {

## ans <- lapply(X = split(e, grp), FUN = FUN, ...)

## if (simplify && length(len <- unique(lengths(ans))) == 1L) {

## if (len == 1L) {

## cl <- lapply(ans, oldClass)

## cl1 <- cl[[1L]]

## ans <- unlist(ans, recursive = FALSE)

## if (!is.null(cl1) && all(vapply(cl, identical, NA,

## y = cl1)))

## class(ans) <- cl1

## }

## else if (len > 1L)

## ans <- matrix(unlist(ans, recursive = FALSE), nrow = nry,

## ncol = len, byrow = TRUE, dimnames = if (!is.null(nms <- names(ans[[1L]])))

## list(NULL, nms))

## }

## ans

## })

##

## [[22]]

## len <- length(y)

##

## [[23]]

## for (i in seq\_along(z)) y[[len + i]] <- z[[i]]

##

## [[24]]

## names(y) <- c(names(by), names(x))

##

## [[25]]

## row.names(y) <- NULL

##

## [[26]]

## y

Now we can choose a point in the function body, where we would like to interactively explore. For example the 21st element starting with z <- lapply(x, function(e)) { may be of interest. In that case, we can call:

trace(stats::aggregate.data.frame, tracer = browser, at = 21)

## Tracing function "aggregate.data.frame" in package "stats"

## [1] "aggregate.data.frame"

And see that this has added a call to .doTrace() to the function body:

as.list(body(stats::aggregate.data.frame))[[21L]]

## {

## .doTrace(browser(), "step 21")

## z <- lapply(x, function(e) {

## ans <- lapply(X = split(e, grp), FUN = FUN, ...)

## if (simplify && length(len <- unique(lengths(ans))) ==

## 1L) {

## if (len == 1L) {

## cl <- lapply(ans, oldClass)

## cl1 <- cl[[1L]]

## ans <- unlist(ans, recursive = FALSE)

## if (!is.null(cl1) && all(vapply(cl, identical,

## NA, y = cl1)))

## class(ans) <- cl1

## }

## else if (len > 1L)

## ans <- matrix(unlist(ans, recursive = FALSE),

## nrow = nry, ncol = len, byrow = TRUE, dimnames = if (!is.null(nms <- names(ans[[1L]])))

## list(NULL, nms))

## }

## ans

## })

## }

When we now call the aggregate() function on a data.frame, we will have the code stop at our selected point in the execution of the data.frame method:

aggregate(mtcars["hp"], mtcars["carb"], FUN = mean)

When done debugging, use untrace() to cancel the tracing:

untrace(stats::aggregate.data.frame)

## Untracing function "aggregate.data.frame" in package "stats"

Happy investigating and debugging!

**References**

**R documentation of the referenced functions**

* R documentation on debug(), debugonce(), etc.
* R documentation on trace(), untrace(), etc.
* R documentation on debugcall()