Diagnosing and fixing errors in your code can be time-consuming and frustrating. There are two ways you can make your life easier. The first is knowing the tools at your disposal in RStudio to debug errors. RStudio provides a variety of tools to help you diagnose the problem at its source and come up with a solution as quick as possible. The second is knowing how to write functions that return clear yet detailed errors using condition handling. This post will walk through both of these topics so that you can become better at handling errors when writing your own code as well as working with errors in someone else’s code.

**Debugging Errors**

The following general strategy can be applied to debug an error, as outlined by [Hadley Wickham in his Advanced R book](https://adv-r.hadley.nz/debugging.html#debugging-strategy):

1. Google
   * Many times it is a common error with a known solution
2. Make it repeatable
   * Create a minimal, reproducible example (e.g. reprex) using simple data
   * Note which inputs don’t trigger the error
   * If not already done, write simple tests to reduce chances of creating a new bug
3. Figure out where the error is
   * Use the “scientific method”
   * Hypothesize, test with experiments, and record results
   * If needed, ask someone else for a second pair of eyes to review
4. Fix it and test it

These four steps should be followed each time you encounter an unexpected error in a function. Many times, you may not even know what line of code the error is coming from. How can you determine where the code is not behaving? You can follow these general steps to answer this question:

1. Begin running the code.
2. Stop the code where you suspect the bug/problem is arising.
3. Look and/or walk through the code, step-by-step at that point.

This can be done ad-hoc in a separate R script containing the function code, or using several built-in tools in RStudio, including the [traceback function](https://www.rdocumentation.org/packages/base/versions/3.6.1/topics/traceback?tap_a=5644-dce66f&tap_s=10907-287229) and [debug mode](https://stat.ethz.ch/R-manual/R-devel/library/base/html/debug.html).

Let’s look at an example function to demonstrate the use of these tools. We’ll create a simple data set with three binary variables, treatment, gender, and outcome. The [chifishr::chi\_fisher\_p function](https://gitlab.com/scheidec/chifishr) is a simple function that calculates a p-value from either a Chi-squared or Fisher Exact test, depending on if a warning is thrown from the Chi-squared test due to small expected counts leading to poor p-value approximations.

treatment <- tibble::tibble(

treatment = c(rep("old", 50), rep("new", 50)),

gender = c(rep("male", 30), rep("female", 20),

rep("male", 20), rep("female", 30)),

outcome = c(rep("failure", 95), rep("success", 5))

)

# devtools::install\_git("<https://gitlab.com/scheidec/chifishr>")

library(chifishr)

# warning is present, Fisher p-value is returned

chi\_fisher\_p(treatment, "outcome", "treatment")

## [1] 0.05628449

# no warning is present, Chi-squared p-value returned

chi\_fisher\_p(treatment, "gender", "treatment")

## [1] 0.07186064

Let’s take a closer look at the code within the chi\_fisher\_p function to see what is happening:

chi\_fisher\_p

## function (tbl, var, treatment)

## {

## chisq\_wrapper <- function(tbl, var, treatment) {

## var <- tbl %>% dplyr::pull(var) %>% as.factor()

## treatment <- tbl %>% dplyr::pull(treatment) %>% as.factor()

## p <- stats::chisq.test(var, treatment)$p.value

## return(p)

## }

## fisher\_wrapper <- function(tbl, var, treatment) {

## var <- tbl %>% dplyr::pull(var) %>% as.factor()

## treatment <- tbl %>% dplyr::pull(treatment) %>% as.factor()

## p <- stats::fisher.test(var, treatment)$p.value

## return(p)

## }

## chisq\_wrapper <- purrr::quietly(chisq\_wrapper)

## chisq <- chisq\_wrapper(tbl, var, treatment)

## if (length(chisq$warnings) == 0) {

## return(chisq$result)

## }

## else {

## return(fisher\_wrapper(tbl, var, treatment))

## }

## }

##

##

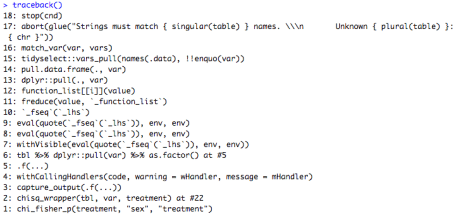
First, there are two internal functions defined, chisq\_wrapper and fisher\_wrapper. These functions pull and store the specified variables from the input tbl as vectors. The chisq.test and fisher.test functions, respectively, are then performed on those vectors and only the numeric p.value result is returned.

The next line wraps the chisq\_wrapper function in [purrr::quietly](https://purrr.tidyverse.org/reference/safely.html), which captures the side effects of a function. Now, when chisq\_wrapper is called, it will return a list with components result, output, messages and warnings. This allows the function to check if a warning is present when the Chi-squared test is performed, and return either the Chi-squared test p-value or the Fisher Exact test p-value from the subsequent if-else block.

If we pass in a variable that is not in the input data set, we would expect an error to be thrown:



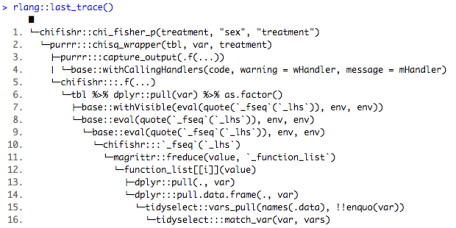
When an error occurs, the interactive “Show Traceback” feature in RStudio (button shown above) or the traceback() function can be very helpful to debug the source of the error. Both options show the call stack that the code runs through before producing the error returned to the user. In many cases there are multiple nested functions that the code uses underneath the top-level function called.



Working from the bottom to the top, we see in the traceback() output call stack that an internal function, chisq\_wrapper() is called within chi\_fisher\_p(). Since the chisq\_wrapper() function was wrapped in purrr::quietly(), the next three functions called, capture\_output(), withCallingHandlers(), and .f(...) are internal functions within purrr::quietly() that are used to capture the warnings and messages output from the chisq\_wrapper() function.

In line 6 of the traceback, we see the code executes line #5 of the chi\_fisher\_p() function, which is the first line of the chisq\_wrapper() function, where it pulls the var variable from the input tbl. This is where the source of the error is. Lines 7-18 are all function calls that are resulting from the dplyr::pull() function and the fact that we are using the pipe, %>% to pass in the inputs to that function. We see in line 17 that the function is trying to pull a variable from the input data frame, but the string passed in does not match a variable name in that data set, which causes the function to abort and return an error.

An alternative to traceback(), rlang::last\_trace() is ordered in the opposite way and shows the hierarchical structure of the call stack:

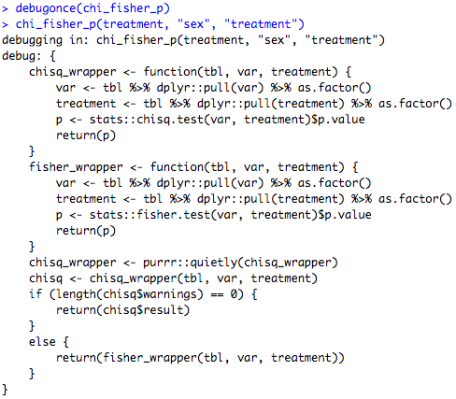


The output is mostly similar to traceback(), but now it is clearer which packages are calling which function since the package prefixes are shown. It is also much easier to see how the nested functions relate to each other. Notice that although the chisq\_wrapper function is not technically a function inside the purrr package, it is evaluated internally (with three colons, :::) from that package due to it being wrapped in purrr::quietly().

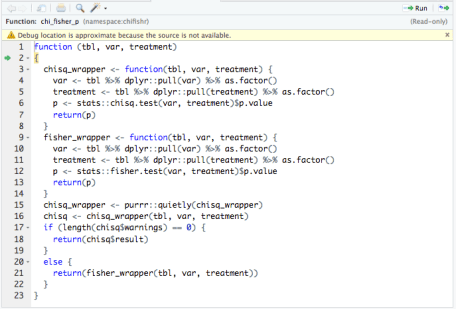
**Interactive Debugger**

If the location of the error from traceback is not enough, try the interactive debugger, which pauses execution of the function and allows you to interactively explore its state. To enter the interactive debugger, you can either use RStudio’s “Rerun with Debug” tool, or use the [debug() or debugonce() functions](https://www.rdocumentation.org/packages/base/versions/3.6.1/topics/debug?tap_a=5644-dce66f&tap_s=10907-287229). The “Rerun with Debug” button pops up when an error occurs, below the “Show Traceback” button shown above in our example error.

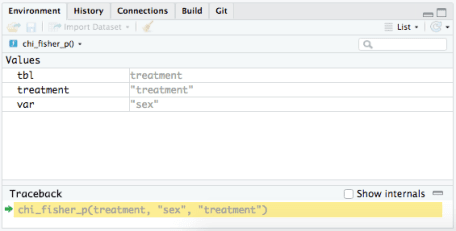
The debugger puts you in an interactive environment *inside* the function where you can run code to explore the current state with the inputs used. You’re in the interactive debugger when you get the special prompt: Browse[1]>. Here, we will pass the debugonce() function to open the interactive debugger only the next time that the chi\_fisher\_p function returns an error and rerun the code that threw the error.



The code for the function you are debugging will open in a new editor window:



Objects in the current environment are now the only objects shown in the Environment pane:



In debug mode, a toolbar containing special commands pops up in the console which can be used in addition to running regular R code:



* *Next*: executes the next line in the function
* *Step into*: works like Next, but if the next step is a function, it will step into that function to be explored interactively.
* *Finish*: finishes execution of the current loop or function
* *Continue*: leaves interactive debugger and continues regular execution of the function. This is useful if you’ve fixed the bad state and want to check that the function correctly executes the remaining code
* *Stop*: stops debugger, terminates the function and returns to the global workspace

In this example, we we can use the *Next* button to step through each line of code within the function and find that the function errors out at the line that calls the chisq\_wrapper function, as we saw before in the traceback call. We could *Step into* that intermediate function and follow through to each consecutive call that the traceback output returned to find the source of the top-level error.

A function may also generate an unexpected warning or message. To diagnose where these are thrown, the easiest way is to convert them to errors using options(warn = 2) for warnings or rlang::with\_abort(function, "message") for messages and then use the debugging tools described above.

**Condition Handling**

Now that we know how to locate and fix problems in R code, we should know the recommended ways of writing functions to communicate problems to other users as clearly as possible. This is the job of [*conditions*](https://adv-r.hadley.nz/conditions.html). There are three types of conditions:

* Errors – raised by stop() or rlang::abort(), force all execution to terminate
* Warnings – raised by warning() or rlang::warn(), display potential problems
* Messages – raised by message() or rlang::inform(), give informative output

Consider an example function, is\_prime() that checks if a number input is a prime number, but raises an error when a negative number is input:

is\_prime <- function(n) {

if((n) < 0) {

stop("n must be positive")

} else {

n == 2L || all(n %% 2L:max(2,floor(sqrt(n))) != 0)

}

}

is\_prime(-3)

## Error in is\_prime(-3): n must be positive

This is a straightforward error, but it could be more detailed, specifically stating what the value of n passed in was.

[tryCatch](https://www.rdocumentation.org/packages/base/versions/3.0.3/topics/conditions?tap_a=5644-dce66f&tap_s=10907-287229) is a way to inspect condition objects and control what happens when a condition is signaled. Let’s define an error handler to decide what happens when is\_prime() fails.

prime\_cnd <- tryCatch(error = function(cnd) cnd, is\_prime(-1))

str(prime\_cnd)

## List of 2

## $ message: chr "n must be positive"

## $ call : language is\_prime(-1)

## - attr(\*, "class")= chr [1:3] "simpleError" "error" "condition"

The resulting object from tryCatch() is a list that contains the condition/error returned as well as the function call that raised the error. We can see that the number passed in, -1, is saved in the call element of the list. This is useful information that can help us in writing a more informative error message.

**Conditions with a chain of functions**

In practice, we usually write functions that call other functions, and it can get confusing if there isn’t an easy way to find the source of the error in nested functions. Let’s define two functions as an example. The first, sim\_value will simulate random input values from a normal(0, 1) distribution, but will raise an error if the returned value is negative. The second function, sqrt\_value will call sim\_value and take the square root of the simulated value.

set.seed(9)

sim\_value <- function(){

val <- rnorm(n = 1, mean = 0, sd = 1)

if (val < 0){

stop("Value returned is negative")

} else {

val

}

}

sqrt\_value <- function(){

x <- sim\_value()

sqrt(x)

}

If val is negative in sim\_value() the same error is thrown in both functions.

sim\_value()

## Error in sim\_value(): Value returned is negative

sqrt\_value()

## Error in sim\_value(): Value returned is negative

Note that the condition returned shows no info about the value of val that caused the error. How can we write more detailed messages when this error is thrown?

**Conditions in rlang**

rlang’s condition functions make it very easy to add the type of custom metadata we want returned in our conditions. To show this, let’s modify sim\_value() to use rlang::abort() instead of stop().

sim\_value <- function(){

val <- rnorm(n = 1, mean = 0, sd = 1)

if (val < 0){

rlang::abort(message = "Value returned is negative",

.subclass ="sim\_value\_error",

val = val)

} else {

val

}

}

Note there are three arguments passed to rlang::abort():

* message: the error message which is similar to the one passed to stop() in the previous example.
* .subclass: a subclass of the condition to differentiate errors.
* val: the particular value that caused the error.

Now use tryCatch() again to inspect the custom condition:

# define an error handler to return the custom error object

custom\_cnd <- tryCatch(error = function(cnd) cnd, sim\_value())

# inspect custom\_cnd

str(custom\_cnd, max.level = 1)

## List of 4

## $ message: chr "Value returned is negative"

## $ trace :List of 4

## ..- attr(\*, "class")= chr "rlang\_trace"

## $ parent : NULL

## $ val : num -0.142

## - attr(\*, "class")= chr [1:4] "sim\_value\_error" "rlang\_error" "error" "condition"

We see the metadata value of val as well as the subclass is captured in the function that uses rlang::abort().

Now we can use this to get a more precise message when we call sqrt\_value() which calls this function, something like this:

“Can’t calculate value because sim\_value() raised an error as val was negative (-1.25)”

We can define an error handler sim\_value\_handler() to access the values returned in the custom error object thrown by sim\_value() and then return a message based on these values.

sim\_value\_handler <- function(cnd) {

msg <- "Can't calculate value"

if (inherits(cnd, "sim\_value\_error")) {

msg <- paste0(msg, " as `val` passed to `sim\_value()` equals (", cnd$val,")")

}

rlang::abort(msg, "sim\_value\_error")

}

Now if we use sim\_value\_handler with sim\_value() inside sqrt\_value(), we can see an example of the modified error message including the value of val that caused the error:

sqrt\_value <- function(){

x <- tryCatch(error = sim\_value\_handler, sim\_value())

sqrt(x)

}



This simple example shows the several advantages to using rlang instead of base functions for condition handling: the ability to store metadata which can be examined by handler, unhandled errors are automatically saved by abort(), and more detailed messages can be output to the end user.