13 Debugging

In Section 2.18, I gave some general advice about what to do when there is an error in your R code:

1. Read the error message carefully and try to decipher it. Have you seen it before? Does it point to a particular variable or function? Check Section 13.2 of this book, which deals with common error messages in R.
2. Check your code. Have you misspelt any variable or function names? Are there missing brackets, strange commas, or invalid characters?
3. Copy the error message and do a web search using the message as your search term. It is more than likely that somebody else has encountered the same problem, and that you can find a solution to it online. This is a great shortcut for finding solutions to your problem. In fact, **this may well be the single most important tip in this entire book**.
4. Read the documentation for the function causing the error message, and look at some examples of how to use it (both in the documentation and online, e.g., in blog posts). Have you used it correctly?
5. Use the debugging tools presented in Chapter 13, or try to simplify the example that you are working with (e.g., removing parts of the analysis or the data) and see if that removes the problem.
6. If you still can't find a solution, post a question at a site like Stack Overflow or the RStudio community forums. Make sure to post your code and describe the context in which the error message appears. If at all possible, post a reproducible example, i.e., a piece of code that others can run, that causes the error message. This will make it a lot easier for others to help you.

The debugging tools mentioned in point 5 above are an important part of your toolbox, particularly if you're doing more advanced programming with R.

In this chapter you will learn how to:

- Debug R code,

- Recognise and resolve common errors in R code, and
- Interpret and resolve common warning messages in R.

13.1 Debugging

Debugging is the process of finding and removing bugs in your scripts. R and RStudio have several functions that can be used for this purpose. We'll have a closer look at some of them here.

13.1.1 Find out where the error occured with `traceback`

If a function returns an error, it is not always clear *where* exactly the error occurred. Let's say that we want to compute the correlation between two variables, but we have forgotten to assign values to the variables:

```
cor(variable1, variable2)
```

The resulting error message is:

```
> cor(variable1, variable2)
Error in is.data.frame(y) : object 'variable2' not found
```

Why is the function `is.data.frame` throwing an error? We were using `cor`, not `is.data.frame`!

Functions often make calls to other functions, which in turn make calls to other functions, and so on. When you get an error message, the error could have taken place in any one of these functions. To find out in which function the error occurred, you can run `traceback`, which shows the sequence of calls that lead to the error:

```
traceback()
```

Which in this case will yield the output:

```
> traceback()
2: is.data.frame(y)
1: cor(variable1, variable2)
```

What this tells you is that `cor` makes a call to `is.data.frame`, and that is where the error occurs. This can help you understand why a function that you weren't aware you were calling (`is.data.frame` in this case) is throwing an error, but won't tell you *why* there was an error. To find out, you can use `debug`, which we'll discuss next.

As a side note, if you'd like to know why and when `cor` called `is.data.frame` you can print the code for `cor` in the Console by typing the function name without parentheses:

```
cor
```

Reading the output, you can see that it makes a call to `is.data.frame` on the 10th line:

```
1  function (x, y = NULL, use = "everything", method = c("pearson",
2      "kendall", "spearman"))
3  {
4      na.method <- pmatch(use, c("all.obs", "complete.obs",
5                                 "pairwise.complete.obs",
6          "everything", "na.or.complete"))
7      if (is.na(na.method))
8          stop("invalid 'use' argument")
9      method <- match.arg(method)
10     if (is.data.frame(y))
11         y <- as.matrix(y)

...
```

13.1.2 Interactive debugging of functions with `debug`

If you are looking for an error in a script, you can simply run the script one line at a time until the error occurs, to find out where the error is. But what if the error is inside of a function, as in the example above?

Once you know in which function the error occurs, you can have a look inside it using `debug`. `debug` takes a function name as input, and the next time you run that function, an interactive debugger starts, allowing you to step through the function one line at a time. That way, you can find out exactly where in the function the error occurs. We'll illustrate its use with a custom function:

```
transform_number <- function(x)
{
  square <- x^2
  if(x >= 0) { logx <- log(x) } else { stop("x must be positive") }
  if(x >= 0) { sqrtx <- sqrt(x) } else { stop("x must be positive") }
  return(c(x.squared = square, log = logx, sqrt = sqrtx))
}
```

The function appears to work just fine:

```
transform_number(2)
transform_number(-1)
```

However, if we input an `NA`, an error occurs:

```
transform_number(NA)
```

We now run `debug`:

```
debug(transform_number)
transform_number(NA)
```

Two things happen. First, a tab with the code for `transform_number` opens. Second, a *browser* is initialised in the Console panel. This allows you to step through the code, by typing one of the following and pressing Enter:

- `n` to run the next line,
- `c` to run the function until it finishes or an error occurs,
- a variable name to see the current value of that variable (useful for checking that variables have the intended values),
- `Q` to quit the browser and stop the debugging.

If you either use `n` a few times, or `c`, you can see that the error occurs on line number 4 of the function:

```
if(x >= 0) { logx <- log(x) } else { stop("x must be positive") }
```

Because this function was so short, you could probably see that already, but for longer and more complex functions, `debug` is an excellent way to find out where exactly the error occurs.

The browser will continue to open for debugging each time `transform_number` is run. To turn it off, use `undebug`:

```
undebug(transform_number)
```

13.1.3 Investigate the environment with `recover`

By default, R prints an error message, returns to the global environment, and stops the execution when an error occurs. You can use `recover` to change this behaviour so that R stays in the environment where the error occurred. This allows you to investigate that environment, e.g., to see if any variables have been assigned the wrong values.

```
transform_number(NA)
recover()
```

This gives you the same list of function calls as `traceback` (called the *function stack*), and you can select which of these you'd like to investigate (in this case there is only one, which you access by writing `1` and pressing Enter). The environment for that call shows up in the Environment panel, which in this case shows you that the local variable `x` has been assigned the value `NA` (which is what causes an error when the condition `x >= 0` is checked).

13.2 Common error messages

Some errors are more frequent than others. Below is a list of some of the most common ones, along with explanations of what they mean, and how to resolve them.

13.2.1 `+`

If there is a `+` sign at the beginning of the last line in the Console, and it seems that your code doesn't run, that is likely due to missing brackets or quotes. Here is an example where a bracket is missing:

```
> 1 + 2*(3 + 2
+
```

Type `)` in the Console to finish the expression, and your code will run. The same problem can occur if a quote is missing:

```
> myString <- "Good things come in threes
+
```

Type `"` in the Console to finish the expression, and your code will run.

13.2.2 `could not find function`

This error message appears when you try to use a function that doesn't exist. Here is an example:

```
> age <- c(28, 48, 47, 71, 22, 80, 48, 30, 31)
> means(age)
Error in means(age) : could not find function "means"
```

This error is either due to a misspelling (in which case you should fix the spelling) or due to attempting to use a function from a package that hasn't been loaded (in which case you should load the package using `library(package_name)`). If you are unsure which package the function belongs to, doing a quick web search for "R function_name" usually does the trick.

13.2.3 `object not found`

R throws this error message if we attempt to use a variable that does not exist:

```
> name_of_a_variable_that_doesnt_exist + 1 * pi^2
Error: object 'name_of_a_variable_that_doesnt_exist' not found
```

This error may be due to a spelling error, so check the spelling of the variable name. It is also commonly encountered if you return to an old R script and try to run just a part of it – if the variable is created on an earlier line that hasn't been run, R won't find it because it hasn't been created yet.

13.2.4 `cannot open the connection` and `No such file or directory`

This error message appears when you try to load a file that doesn't exist:

```
> read.csv("not-a-real-file-name.csv")
Error in file(file, "rt") : cannot open the connection
In addition: Warning message:
In file(file, "rt") :
  cannot open file 'not-a-real-file-name.csv': No such file or
  directory
```

Check the spelling of the file name, and that you have given the correct path to it (see Section 2.15). If you are unsure about the path, you can use

```
read.csv(file.choose())
```

to interactively search for the file in question.

13.2.5 invalid 'description' argument

When you try to import data from an Excel file, you can run into error messages like:

```
Error in file(con, "r") : invalid 'description' argument
In addition: Warning message:
In unzip(xlsxFile, exdir = xmlDir) : error 1 in extracting from zip file
```

and

```
Error: Evaluation error: zip file 'C:\Users\mans\Data\some_file.xlsx' cannot be
opened.
```

These usually appear if you have the file open in Excel *at the same time* that you're trying to import data from it in R. Excel temporarily locks the file so that R can't open it. Close Excel and then import the data.

13.2.6 missing value where TRUE/FALSE needed

This message appears when a condition in a conditional statement evaluates to `NA`. Here is an example:

```
x <- c(8, 5, 9, NA)
for(i in seq_along(x))
{
    if(x[i] > 7) { cat(i, "\n") }
}
```

which yields:

```
> x <- c(8, 5, 9, NA)
> for(i in seq_along(x))
+ {
+     if(x[i] > 7) { cat(i, "\n") }
+ }
1
3
Error in if (x[i] > 7) { : missing value where TRUE/FALSE needed
```

The error occurs when `i` is `4`, because the expression `x[i] > 7` becomes `NA > 7`, which evaluates to `NA`. `if` statements require that the condition evaluates to either `TRUE` or `FALSE`. When this error occurs, you should investigate why you get an `NA` instead.

13.2.7 `unexpected '=' in ...`

This message indicates that you have an assignment happening in the wrong place. You probably meant to use `==` to check for equality, but accidentally wrote `=` instead, as in this example:

```
x <- c(8, 5, 9, NA)
for(i in seq_along(x))
{
    if(x[i] = 5) { cat(i, "\n") }
}
```

which yields:

```
> x <- c(8, 5, 9, NA)
> for(i in seq_along(x))
+ {
+     if(x[i] = 5) { cat(i, "\n") }
Error: unexpected '=' in:
"{
    if(x[i] ="
> }
Error: unexpected '}' in "}"
```

Replace the `=` by `==` and your code should run as intended. If you really intended to assign a value to a variable inside the `if` condition, you should probably rethink that.

13.2.8 `attempt to apply non-function`

This error occurs when you put parentheses after something that isn't a function. It is easy to make that mistake, e.g., when doing a mathematical computation.

```
> 1+2(2+3)
Error: attempt to apply non-function
```

In this case, we need to put a multiplication symbol `*` between `2` and `(` to make the code run:

```
> 1+2*(2+3)
[1] 11
```

13.2.9 `undefined columns selected`

If you try to select a column that doesn't exist from a data frame, this message will be printed. Let's start by defining an example data frame:

```
age <- c(28, 48, 47, 71, 22, 80, 48, 30, 31)
purchase <- c(20, 59, 2, 12, 22, 160, 34, 34, 29)
bookstore <- data.frame(age, purchase)
```

If we attempt to access the third column of the data, we get the error message:

```
> bookstore[,3]
Error in `[.data.frame`(bookstore, , 3) : undefined columns selected
```

Check that you really have the correct column number. It is common to get this error if you have removed columns from your data.

13.2.10 `subscript out of bounds`

This error message is similar to the last example above but occurs if you try to access the column in another way:

```
> bookstore[[3]]
Error in .subset2(x, i, exact = exact) : subscript out of bounds
```

Check that you really have the correct column number. It is common to get this error if you have removed columns from your data, or if you are running a for loop accessing element `[i, j]` of your data frame, where either `i` or `j` is greater than the number of rows and columns of your data.

13.2.11 `Object of type 'closure' is not subsettable`

This error occurs when you use square brackets `[ ]` directly after a function:

```
> x <- c(8, 5, 9, NA)
> sqrt[x]
Error in sqrt[x] : object of type 'closure' is not subsettable
```

You probably meant to use parentheses `( )` instead. Or perhaps you wanted to use the square brackets on the object returned by the function:

```
> sqrt(x)[2]
[1] 2.236068
```

13.2.12 `$ operator is invalid for atomic vectors`

This message is printed when you try to use the `$` operator with an object that isn't a `list` or a data frame, for instance with a vector. Even though the elements in a vector can be named, you cannot access them using `$`:

```
> x <- c(a = 2, b = 3)
> x
a b
2 3
> x$a
Error in x$a : $ operator is invalid for atomic vectors
```

If you need to access the element named `a`, you can do so using bracket notation:

```
> x["a"]
a
2
```

Or use a data frame instead:

```
> x <- data.frame(a = 2, b = 3)
> x$a
[1] 2
```

13.2.13 `(list) object cannot be coerced to type 'double'`

This error occurs when you try to convert the elements of a `list` to `numeric`. First, we create a `list`:

```
x <- list(a = c("1", "2", "3"),
          b = c("1", "4", "1889"))
```

If we now try to apply `as.numeric`, we get the error:

```
> as.numeric(x)
Error: 'list' object cannot be coerced to type 'double'
```

You can apply `unlist` to collapse the `list` to a vector:

```
as.numeric(unlist(x))
```

You can also use `lapply` (see Section 6.5):

```
lapply(x, as.numeric)
```

13.2.14 arguments imply differing number of rows

This message is printed when you try to create a data frame with different numbers of rows for different columns, like in this example, where `a` has three rows and `b` has four:

```
> x <- data.frame(a = 1:3, b = 6:9)
Error in data.frame(a = 1:3, b = 6:9) :
  arguments imply differing number of rows: 3, 4
```

If you really need to create an object with different numbers of rows for different columns, create a `list` instead:

```
x <- list(a = 1:3, b = 6:9)
```

13.2.15 non-numeric argument to a binary operator

This error occurs when you try to use mathematical operators with non-numerical variables. For instance, it occurs if you try to add `character` variables:

```
> "Hello" + "World"
Error in "Hello" + "world" : non-numeric argument to binary operator
```

If you want to combine `character` variables, use `paste` instead:

```
paste("Hello", "world")
```

13.2.16 non-numeric argument to mathematical function

This error message is similar to the previous one, and it appears when you try to apply a mathematical function, like `log` or `exp` to non-numerical variables:

```
> log("1")
Error in log("1") : non-numeric argument to mathematical function
```

Make sure that the data you are inputting doesn't contain `character` variables.

13.2.17 `cannot allocate vector of size ...`

This message is shown when you're trying to create an object that would require more RAM than is available. You can try to free up RAM by closing other programs and removing data that you don't need using `rm` (see Section 5.14). Also check your code so that you don't make copies of your data, which takes up more RAM. Replacing base R and `dplyr` code for data wrangling with `data.table` code can also help, as `data.table` uses considerably less RAM for most tasks.

13.2.18 `Error in plot.new() : figure margins too large`

This error occurs when your Plot panel (or file, if you are saving your plot as a graphics file) is too small to fit the graphic that you're trying to create. Enlarge your Plot panel (or increase the size of the graphics file) and run the code again.

13.2.19 `Error in .Call.graphics(C_palette2, .Call(C_palette2, NULL)) : invalid graphics state`

This error can happen when you create plots with `ggplot2`. You can usually solve it by running `dev.off()` to close the previous plot window. In rare cases, you may have to reinstall `ggplot2` (see Section 12.1).

13.2.20 `Error in select(...) : unused argument (...)`

This error occurs when you attempt to use `select` from `dplyr` after loading the `MASS` package, which also has a function called `select`. To solve it, specify that you wish to use `select` from `dplyr`:

```
airquality |> dplyr::select(Ozone)
```

13.3 Common warning messages

13.3.1 `replacement has ... rows ...`

This occurs when you try to assign values to rows in a data frame, but the object you are assigning to them has a different number of rows. Here is an example:

```
> x <- data.frame(a = 1:3, b = 6:8)
> y <- data.frame(a = 4:5, b = 10:11)
> x[3,] <- y
Warning messages:
1: In `[<-.data.frame`(`*tmp*`, 3, , value = list(a = 4:5, b = 10:11)) :
  replacement element 1 has 2 rows to replace 1 rows
2: In `[<-.data.frame`(`*tmp*`, 3, , value = list(a = 4:5, b = 10:11)) :
  replacement element 2 has 2 rows to replace 1 rows
```

You can fix this, e.g., by changing the number of rows to place the data in:

```
x[3:4,] <- y
```

13.3.2 `the condition has length > 1 and only the first element will be used`

This warning is thrown when the condition in a conditional statement is a vector rather than a single value. Here is an example:

```
> x <- 1:3
> if(x == 2) { cat("Two!") }
Warning message:
In if (x == 2) { :
  the condition has length > 1 and only the first element will be used
```

Only the first element of the vector is used for evaluating the condition. See if you can change the condition so that it doesn't evaluate to a vector. If you actually want to evaluate the condition for all elements of the vector, either collapse it using `any` or `all` or wrap it in a

loop:

```
x <- 1:3
if(any(x == 2)) { cat("Two!") }

for(i in seq_along(x))
{
    if(x[i] == 2) { cat("Two!") }
}
```

13.3.3 number of items to replace is not a multiple of replacement length

This error occurs when you try to assign too many values to too short a vector. Here is an example:

```
> x <- c(8, 5, 9, NA)
> x[4] <- c(5, 7)
Warning message:
In x[4] <- c(5, 7) :
  number of items to replace is not a multiple of replacement length
```

Don't try to squeeze more values than can fit into a single element! Instead, do something like this:

```
x[4:5] <- c(5, 7)
```

13.3.4 longer object length is not a multiple of shorter object length

This warning is printed, e.g., when you try to add two vectors of different lengths together. If you add two vectors of equal lengths, everything is fine:

```
a <- c(1, 2, 3)
b <- c(4, 5, 6)
a + b
```

R does *element-wise* addition, i.e., adds the first element of `a` to the first element of `b`, and so on.

But what happens if we try to add two vectors of different lengths together?

```
a <- c(1, 2, 3)
b <- c(4, 5, 6, 7)
a + b
```

This yields the following warning message:

```
> a + b
[1] 5 7 9 8
Warning message:
In a + b : longer object length is not a multiple of shorter object length
```

R *recycles* the numbers in `a` in the addition, so that the first element of `a` is added to the fourth element of `b`. Was that really what you wanted? Maybe. But probably not.

13.3.5 `NAs introduced by coercion`

This warning is thrown when you try to convert something that cannot be converted to another data type:

```
> as.numeric("two")
[1] NA
Warning message:
NAs introduced by coercion
```

You can try using `gsub` to manually replace values instead:

```
x <- c("one", "two")
x <- gsub("one", 1, x)
as.numeric(x)
```

13.3.6 `package is not available (for R version x.x.x)`

This warning message (which perhaps should be an error message rather than a warning) occurs when you try to install a package that isn't available for the version of R that you are using.

```
> install.packages("great_name_for_a_package")
Installing package into '/home/mans/R/x86_64-pc-linux-gnu-library/4.4'
(as 'lib' is unspecified)
Warning in install.packages :
  package 'great_name_for_a_package' is not available (for R version
  4.4.0)
```

This can be either due to the fact that you've misspelt the package name or that the package isn't available for your version of R, either because you are using an out-of-date version or because the package was developed for an older version of R. In the former case, consider updating to a newer version of R. In the latter case, if you really need the package, you can find and download older version of R at R-project.org.

On Windows, it is relatively easy to have multiple versions of R installed side-by-side (in RStudio, you can choose which version to use through the menus, by going to *Tools* > *Global options* and choosing *R version*. On a Mac, you can use the R Switch app available from: https://rud.is/rswitch/