fixURcode

Anna Lohmann

2020-06-10

Contents

Chapter 1

eat(), sleep(), debug(), repeat()

Chapter 2

Error messages

The very obvious first step when your code is not working: Read the error message. If that does not help. *Read it again.* Often you can also google the error message. If you do so try eliminating the parts that seem unique to your own code (e.g. specific dimensions, object names,...)

Pro Tip (for new or niche packages):

If google won't find the error try *github.* You might find the error in the package code or in a github issue.

2.1 Decifer the error message

Asking yourself the following question can help decifer the error message:

- What object is the error message referring to?
- What property of the object might be causing the error?
  Is the object
  - ... missing?
  - ... of the wrong type?
  - ... the wrong size?
  - ... wrong content?
  - ... misspelled?

2.2 Code doesn't work without errors

2.2.1 Raise warnings or messages to errors

You can turn warnings and messages into errors with the foolowing code: `options(warn = 2)`

Promote messages to errors:
`rlang::with_abort(f(), "message")`

2.2.2 No visible warnings or messages

If the malfunctioning code does not produce any errors or warnings check the documentation to see whether some warnings or error messages are turned off by default. Often this is some form of `print = FALSE`, `verbose = FALSE` or `silent = TRUE`.

Chapter 3

Don't assume anything!

- Check the variable or file names.
- Check the object content and type.
- Check code that you think was working fine just a minute ago and you are certain you haven't touched.
- Check whether the online example you modeled your code after actually works.
- Check the function documentation, whether it really works the way you thought it would.
- Write a test to confirm expected behaviour.

Chapter 4

Read the documentation

Reading documentation is a learned skill. It doesn't come naturally and take a lot of time to master. This is not always the user's fault. Also writing documentation is a skill that not all package developers master equally well.

Try always looking for the following elements when reading documentations.

(1) What arguments does the function take?
(2) Which of those arguments have to be provided?
(3) If they don't have to be provided is the default what I want?
(4) Which object types and dimensions should the input have?
(5) Does the input have to be named?

If the documentation is unclear in these regards try inferring some of the information from examples or a package vignette.

If the information on a specific package is scarce. Try the github search to see it in action.

Chapter 5

Most common error sources

In no particular order these are the most important reasons something goes wrong in my code:

- The object is of the wrong type (i.e. a matrix instead of a data.frame or the other way round)
- The object should have names but doesn't.
- I forgot to turn of `stringsaAsFactors = TRUE` (Problem of the past with R 4.0).
- Some object does not exist (typo or silent error in earlier code).
- A `(`, `{`, `,`, `"` is missing or misplaced.
- An object from the environment is used instead of a function parameter.
- The object you are looking for is at a different level of a nested list.
- The method can't handle missing data.
- A namespace conflict calls the wrong function.

Tipp: When copy & paste stuff paste it in a text editor first to avoid weird non-code symbols.

Chapter 6

Check the function code

If you have consulted the documentation and the internet and still have no idea why your code is not doing what you expect it to do it can be helpful to check the function definition.

You can get the code of a given function in different ways. - Click on the function name while pressing shift. - Typing an empty function call in the console

There are a few things that have to be kept in mind: (1) The package containing the function has to be loaded. (2) The function you would like to explore is exported by that package. (3) It is not a generic method.

(2) can be solved by adding `packagename::` in front of the function call. That way you can also explore functions that are not exported by a package.

(3) Can be solved by addding the class in front of the generic function. So if you would like to explore the plot method of the bootnet package, you would have to call it as `plot.bootnet()` in order so see the definition.

6.1 S3 and Debugging

- `UseMethod` errors
- The corresponding functions are called `<function.class>` or
- `<function.default>`
- Use methods() to see what methods are associated with a function or class

Press ctrl + left mouse click on the function name to view the function definition

```
reverseLetters()
```

Chapter 7

Object properties

`str()`

`dplyr::glimpse()`

`class()`

`typeof()`

`names()`

`length()`

`dim()`

`print()`

`View()`

`head()`

`identical()`

Chapter 8

Debugging strategies

8.1 Isolate the problem

8.1.1 Minimum working example

- Remove as much code as possible but have the problem still occurring.
- Locate the source of the error.
- Change one thing at a time.

8.2 Line-by-line debugging (Stepping through the code)

- Run the code line by line

8.3 Rubber duck debugging

Talk yourself through the code. (Or explain it to your rubber duck.)

8.4 Print debugging

The idea of print debugging is to print certain objects or object properties close to where you expect the error might occur. Printing might show you that an object has differetn properties from what you would expect it to have or figure our in which iteration or with which seed something might break. This is a quick and dirty option to the more elaborate debugging tools.

8.5 Logging

Chapter 9

Debugging tools

9.0.1 Interactive R session

- R session reacts to user input
- Check with `interactive()`

9.0.2 Step through the code - `debug()`

- `debug(my_defunct_function)`
- Call exported functions from a package with `::`
- Unexported functions can be referred to with `:::`

R will switch to the browser mode every time that function is called from anywhere in R.

You have to tell R to stop debugging `undebug(my_defunct_function)` or overwrite the function (e.g. by sourcing it). `debugonce(my_defunct_function)` will go in browse mode the next time the function is called.

9.0.3 browser()

- Enter browse() anywhere in your code where you want the execution to halt.
- You can then inspect the environment at that state.
- You can also change objects and continue.
- You can also use `browser()` as an error handler.

`Browse[number]` The number tells you at which level of the call stack you are.

9.0.4 Breakpoints

- Does the same as “browse()” but without lottering your code.
- Click left of a line number for a dot to appear.
- Next time you run the code it will stop there.
- `options(error = recover)` automatically switches to browsing when an error occurs

9.0.5 traceback()

- Helps locating the problem.
- Shows you the last commands that have been called before the error occured (i.e. the stack).
- Reverse order, latest command is on top.

`rlang::last_trace()` is ordered in the opposite way to `traceback()`

9.1 Debugging other people’s code

Chapter 10

Ask a peer

Sometimes you are so entangled in your code that you cant see the forest for the trees. What has kept you stuck for days can often be solved within minutes by a fresh pair of eyes. And even if the peer can't sove your problem. The process of walking someone else through it might make you come up with a solution yourself.

Don't assume your error is embarrassing. If it is, your peer will find and fix it within seconds. If it isn't you can both learn from it.

Chapter 11

It works again? - Great, do you know why?

- A random stacl overflow post fixed it?
- You made some random changes?

Take the time to figure out why these fixes work and try making a note what the solution was. Chances are you'll break it again or run into the same error later in your code.

Chapter 12

Use Git!

- Create a bugfix branch (or two or three if you try more complex approaches)
- You can easily get back to your starting point
- You don't create new problems by forgetting to change something back in some remote part of the code that turned out to not work
- You can keep track of what you already tried
- If an attempt almost got you there and you discover the one thing that was missing three hours later you can easily recreate that "95% good" state
- When you fixed the problem you can analyse the solution by looking at what exactly was changed

Chapter 13

Error handling

Buiding error handling into your own code can get messy really fast. Just returning an error message or continuing a loop even when one iteration fails is easy. But as soon as you want to adapt behaviour depending on error or save the error messages, things get messy.

This is not always your fault, but sometimes due to messy error documentation of the package developer.

Chapter 14

Unit tests with `testthat`

Idea: - You are working with a package - The test live in a separate folder `tests/testthat/` - Testing is part of your workflow.

Easy examples for tests: - `expect_error()` the code should throw an error - `expect_warning()` the code should throw a warning - `expect_equal(a, b)` a and b should be the same (up to numeric tolerance) - `expect_equivalent(a, b)` a and b should contain the same values but may have different attributes (e.g., names and dimnames)

If you are not working with a package check out:

Chapter 15

Resources

15.0.1 Debugging with RStudio - Jonathan McPherson

https://support.rstudio.com/hc/en-us/articles/205612627-Debugging-with-RStudio

15.0.2 Advanced R - Hadley Wickham (Chapter 22 Debugging)

https://adv-r.hadley.nz/debugging.html

15.0.3 What They Forgot to Teach You About R - Jennifer Bryan, Jim Hester

https://rstats.wtf/debugging-r-code.html

15.0.4 Toy examples to test your debugging skills

rstd.io/wtf-debugging

15.0.5 Error handling

15.0.6 Unit tests

Advanced R - Hadley Wickham (Chapter Tests) http://r-pkgs.had.co.nz/tests.html

Chapter 16

Getting frustrated?

- Are you trying the same thing for the third time?
- You can't remember which of the 20 open tabs you have already tried?
- Hitting your computer or cursing at the screen?
- Randomly changing stuff?
- Copy and paste cryptic code snippets from stack overflow?
- Take a deep breath!
- Remind yourself that computers are completely rational. Noone is out there to get you! There is a logical reason! It didn't just magically break!
- Take a break!
- Be determined to understand the problem.
- Embrace the challenge and the learning experience!

Chapter 17

Buggy code to test

```
usethis::use_course("rstd.io/wtf-debugging")
```