

Lecture 09 – Debugging

Learning Objectives:

1. Become proficient in the use of the R language

1.8 Learn the basics of debugging interactively and non-interactively in R.

3. Learn the basic principles of software design.

3.6 Learn effective strategies for debugging code interactively and non-interactively.

3.7 Practice defensive coding.

Debugging strategy

Hadley Wickham's Steps for Debugging (from *Advanced R*)

1. **Google it!** Googling will sometimes yield a solution.

- Remove variable and function names
- Read and understand the post, why the error occurs, and why the fix works.
- Read a variety of posts to see if the fixes are consistent or if there are other options.

2. **Make it repeatable.** Form a hypothesis and test it until it works out.

- Copy code into separate script and run it that way.
- Create an automated test.

3. **Find out where it is.** Critical step to find where in the script the error originates.

- Use `traceback()` to find error
- For functions, define parameters manually and run line-by-line
- For for loops, define your looped parameter and run line-by-line
- There are tools!

4. **Fix it and test it.** Be sure to make the error a test!

Tools that will help: RStudio's debugger

Visualizing problems

traceback()

The screenshot displays the RStudio interface with the debugger active. The top-left pane shows the source code of the `plotahair` function. The top-right pane shows the `Traceback` window, which lists the sequence of function calls leading to the error. The bottom-left pane shows the console with a stack trace of the error.

Source Code (plotahair):

```
function(hairxCenterx, hairxCentery, hdia, dx, no, plotit){  
  h1 <- circle(c(hairxCenterx, hairxCentery), 0.5 * hdia, "dx")  
  if(plotit == 1){  
    points(hairxCenterx, hairxCentery, pch = 19, cex = 2.5)  
    text(hairxCenterx, hairxCentery, labels = no, col = "red")  
  }  
  return(h1)  
}
```

Traceback:

```
eval(substitute(browse(skipCalls = skip), list(skip = 7 ...  
eval(substitute(browse(skipCalls = skip), list(skip = 7 ...  
(function () at post-refactoring_generate_grid2d.R:0  
seq.default(-(radius + 0.01), radius + 0.01, by = dx) at  
seq.default(-(radius + 0.01), radius + 0.01, by = dx) at
```

Console:

```
Error in del/by : non-numeric argument to binary operator  
Enter a frame number, or 0 to exit  
1: source("~/Dropbox (Chapman)/courses/CS510/CourseInfoFall2020/LectureNotes/12-Refactor/Refactoring_example/  
2: withVisible(eval(ei, envir))  
3: eval(ei, envir)  
4: eval(ei, envir)  
5: post-refactoring_generate_grid2d.R#177: makehairs(parameters$angle[j], parameters$gap[j], j, nohairs, pl  
6: post-refactoring_generate_grid2d.R#137: plotahair(hairx, hairy, hdia, dx, i, plotit)  
7: post-refactoring_generate_grid2d.R#69: circle(c(hairxCenterx, hairxCentery), 0.5 * hdia, "dx")  
8: post-refactoring_generate_grid2d.R#41: seq(-(radius + 0.01), radius + 0.01, by = dx)  
9: seq.default(-(radius + 0.01), radius + 0.01, by = dx)  
Selection: 6  
Called from: eval(substitute(browse(skipCalls = skip), list(skip = 7 - which)),  
envir = sys.frame(which))  
Browse[1]> |
```

Debugger functions

Non-interactive debugging

Stick with the general strategy, but there are fewer tools to help.

Check common problems:

- Did you run the code in a clean session?
- Is the working directory different?
- Is the path environment different?
- Is the R_LIBS path different?

Use `dump.frames()` (like `recover()` in interactive mode):

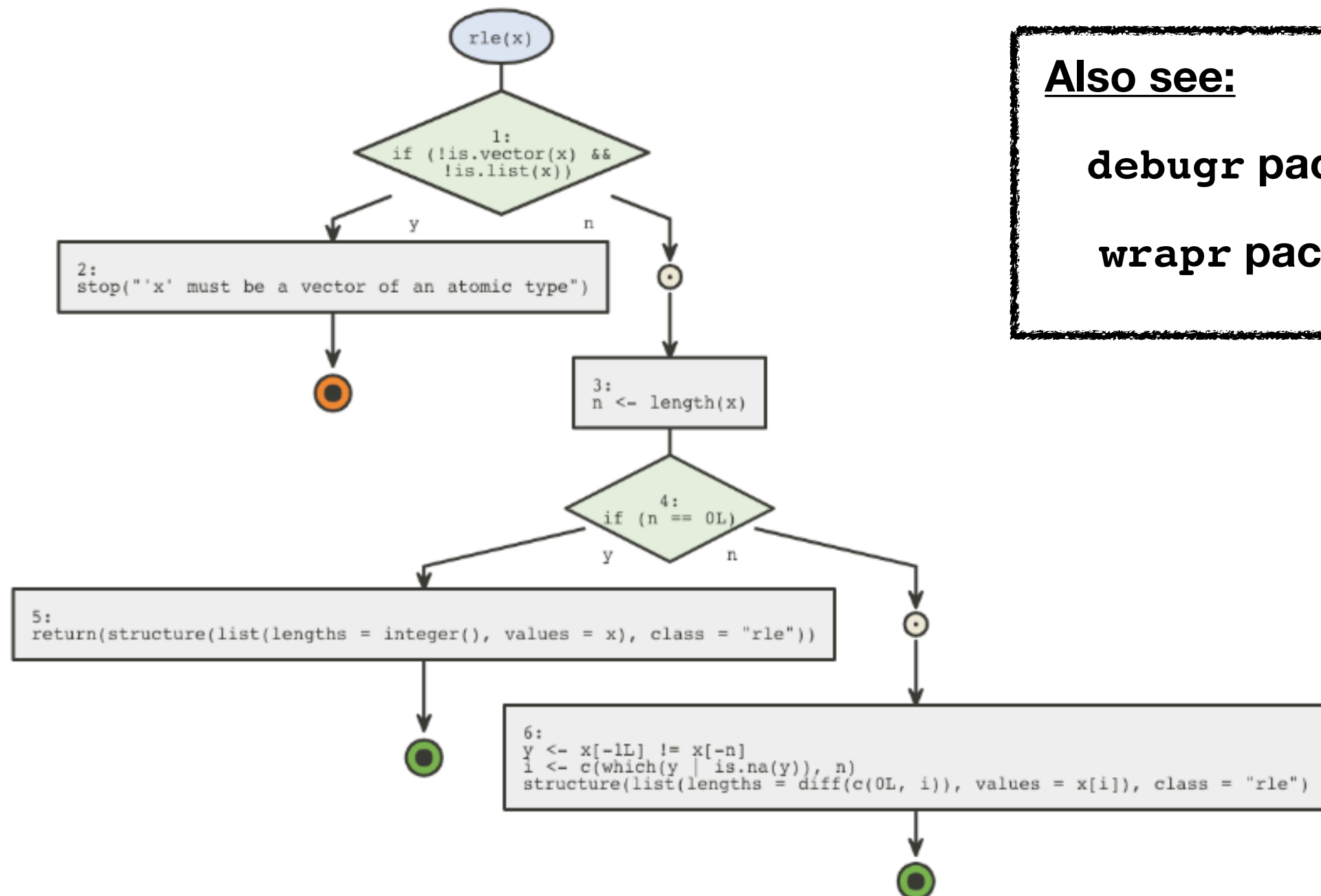
```
dump_and_quit <- function() {  
  # Save debugging info to file las.dump.rda  
  dump.frames(to.file = TRUE)  
  # Quits R with error status  
  q(status = 1)  
}  
  
options(error = dump_and_quit)
```

Tools that will help: flow package

To install: `remotes::install_github("moodymudskipper/flow")`

Draws flow charts of functions. Useful for a deep dive into analysis pipelines.

<https://moodymudskipper.github.io/flow/articles/Draw-a-function.html>



Also see:

debugr package

wrapr package

Defensive programming

“Fail fast.”

- Be strict about what you accept.
- Avoid functions that use non-standard evaluation.
- Avoid functions that return different types of output depending on their input.

Condition handling:

- `try()`
- `tryCatch()`

Additional Resources

**<https://adv-r.hadley.nz/debugging.html#non-interactive-debugging> –
Debugging, *Advanced R***

**<https://data-flair.training/blogs/debugging-in-r-programming/> –
R Debug Essentials**

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

<https://rdr.io/cran/wrapr/f/vignettes/DebugFnW.Rmd> – wrapr Debug Vignette

**<https://cran.r-project.org/web/packages/debugr/vignettes/debugr.html> –
debugr package vignette**