

ETC4500/ETC5450

Advanced R programming

Week 4: Debugging and profiling



Outline

1 Debugging

2 Styling

3 Profiling

4 Efficiency

Outline

1 Debugging

2 Styling

3 Profiling

4 Efficiency

Outline

1 Debugging

2 Styling

3 Profiling

4 Efficiency

What's a bug?

An incorrect, unexpected, or unintended behaviour of code.

💡 Why do we call it a bug?

Why not a mistake? A glitch? An oopsie-daisy?

What's a bug?

On September 9, 1947, a real moth was found causing a malfunction in the Harvard Mark II computer. This incident was recorded in the logbook with the note "First actual case of bug being found."



Overall debugging strategy

Ask for help

- Ask an LLM (OpenAI, Claude, ...)
- Ask a search engine (Google, Bing, DuckDuckGo, ...)
- Ask the community (Stack Overflow / Posit Community, ...)

Fix it yourself

- Update your software / R packages
- Create a minimal reproducible example
- Explore code to find where the error is
- Create a unit tests with expected behaviour
- Fix and test it

Debugging tools in R

- `traceback`: prints out the function call stack after an error occurs; does nothing if there's no error.
- `debug`: flags a function for “debug” mode which allows you to step through execution of a function one line at a time.
- `undebug`: removes the “debug” flag from a function.
- `browser`: pauses execution of a function and puts the function in debug mode.
- `trace`: allows you to insert code into a function at a specific line number.
- `untrace`: removes the code inserted by `trace`.
- `recover`: allows you to modify the error behaviour so that you can browse the function call stack after an error occurs.

Traceback

```
f <- function(a) g(a)
g <- function(b) h(b)
h <- function(c) i(c)
i <- function(d) {
  if (!is.numeric(d)) stop("`d` must be numeric", call. = FALSE)
  d + 10
}
```

```
> f("a")
```

```
Error: `d` must be numeric
```

 Show Traceback

 Rerun with Debug

Traceback

```
f <- function(a) g(a)
g <- function(b) h(b)
h <- function(c) i(c)
i <- function(d) {
  if (!is.numeric(d)) stop("`d` must be numeric", call. = FALSE)
  d + 10
}
```

```
> f("a")
```

```
Error: `d` must be numeric
```

 Hide Traceback

 Rerun with Debug

```
5. stop("`d` must be numeric", call. = FALSE) at debugging.R#6
4. i(c) at debugging.R#3
3. h(b) at debugging.R#2
2. g(a) at debugging.R#1
1. f("a")
```

Traceback

```
f <- function(a) g(a)
g <- function(b) h(b)
h <- function(c) i(c)
i <- function(d) {
  if (!is.numeric(d)) stop("`d` must be numeric", call. = FALSE)
  d + 10
}
```

```
f("a")
#> Error: `d` must be numeric
traceback()
#> 5: stop("`d` must be numeric", call. = FALSE) at debugging.R#6
#> 4: i(c) at debugging.R#3
#> 3: h(b) at debugging.R#2
#> 2: g(a) at debugging.R#1
#> 1: f("a")
```

Interactive debugging

■ Using `browser()`

```
i <- function(d) {  
  browser()  
  if (!is.numeric(d)) stop("`d` must be numeric", call. = FALSE)  
  d + 10  
}
```

■ Setting breakpoints

- ▶ Similar to `browser()` but no change to source code.
- ▶ Set in RStudio by clicking to left of line number, or pressing `Shift+F9`.

■ `options(error = browser)`

Interactive debugging

Debugging commands:

- 1 **n**: Next line (step over).
- 2 **s**: Step into function.
- 3 **c**: Continue to next breakpoint.
- 4 **f**: Finish the current function.
- 5 **q**: Quit debugging.
- 6 **where**: Show the call stack.
- 7 **help**: Help with these debugging commands.

Interactive debugging

- `debug()` : inserts a `browser()` statement at start of function.
- `undebug()` : removes `browser()` statement.
- `debugonce()` : same as `debug()`, but removes `browser()` after first run.

Let's fix a real, unsolved bug.

[#mitchelloharawild/distributional/issues/133](#)

```
distributional::dist_normal() * 2  
#> Error in .mapply(get(op), list(x = vec_data(x), y = y)): argument "MoreArgs" is m
```

The debugging workflow

- 1 Create a reпреx that demonstrates the problem as a comment in the issue.
- 2 Fix the problem in the package code.
- 3 Add a comment to the issue explaining the bug and the fix, including a link to the commit containing the fix.
- 4 Add unit test(s) to the package that confirms the problem is fixed.
- 5 Close the issue.

1 What's wrong with this code?

```
# Multivariate scaling function
mvscale <- function(object) {
  # Remove centers
  mat <- sweep(object, 2L, colMeans(object))
  # Scale and rotate
  S <- var(mat)
  U <- chol(solve(S))
  z <- mat %*% t(U)
  # Return orthogonalized data
  return(z)
}
mvscale(mtcars)
```

Error in `mat %*% t(U)`: requires numeric/complex matrix/vector arguments

Example



Common error messages

- could not find function "xxxx"
- object xxxx not found
- cannot open the connection / No such file or directory
- missing value where TRUE / FALSE needed
- unexpected = in "xxxx"
- attempt to apply non-function
- undefined columns selected
- subscript out of bounds
- object of type 'closure' is not subsettable
- \$ operator is invalid for atomic vectors
- list object cannot be coerced to type 'double'
- arguments imply differing number of rows
- non-numeric argument to binary operator

Common warning messages

- NAs introduced by coercion
- replacement has xx rows to replace yy rows
- number of items to replace is not a multiple of replacement length
- the condition has length > 1 and only the first element will be used
- longer object length is not a multiple of shorter object length
- package is not available for R version xx

Asking for help

To get useful help, it is important that you ask a **good question**. Consider answering these two equivalent questions, which is easier to understand and why?

Asking for help

! urgent help needed with assignment error

My code doesn't work. Please help i need it working for my assignment asap!

```
data <- read.csv("C://Users/James/Downloads/project-a9j-  
2020a/files/survey_data.csv") data %>% filter(y == "A") %>%  
ggplot(aes(y = y, x = temperature)) + geom_line()
```

Asking for help

💡 Error with dplyr filter(): “object not found”

I'm trying to filter a dataset in dplyr, but I'm getting an error that I don't understand. Here's my code and error message:

```
survey <- data.frame(x = c(1, 2, 3), y = c("A", "B", "C"))  
survey %>% filter(y == "A")
```

Error: Error in filter(y == "A") : object 'y' not found

I expected it to return rows where y is "A". How should I fix this?

Minimal reproducible examples

A minimal reproducible example (MRE) is essential for effectively communicating problems with code.

The process of creating a MRE might also help you resolve the problem yourself!

Minimal reproducible examples

Minimal

Minimising code and data makes it easier to find the problem.

- **Remove unnecessary code**

Include as little code as possible to show the problem.

- **Use small datasets**

Prefer built-in datasets or small example datasets.

- **Avoid external dependencies**

Remove unused packages or files irrelevant to the problem

Minimal reproducible examples

Reproducible

■ Required packages

If external packages are needed, include loading the packages in your MRE.

■ Used datasets

If you can't use built-in datasets, provide a minimal dataset with `data.frame()` or `dput()`.

■ Set random seeds

If your problem includes randomisation, include

Minimal reproducible examples

Examples

- **Clearly state the issue**

Explain what you expect versus what happens.

- **Ensure clarity**

Add code comments to highlight your intention and the problem.

The **reprex** package helps create *minimal reproducible examples*.

- Results are saved to clipboard in form that can be pasted into a GitHub issue, Stack Overflow question, or email.
- `reprex::reprex()`: takes R code and outputs it in a markdown format.
- Append session info with `reprex(..., session_info = TRUE)`.
- Use the RStudio addin.

reprex as a debugging tool

Creating increasingly minimal reproducible examples can be a useful debugging tool.

Let's look at this bug:

[#tidyverts/fabletools/issues/350](#)

```
library(fpp3)
us_change %>%
  pivot_longer(c(Consumption, Income), names_to = "Time Series") %>%
  autoplot(value)
#> Error in `not_tsibble()`:
#> ! x is not a tsibble.
```

Exercises

Create a Minimal Reproducible Example (MRE) for this code:

```
library(tidyverse)
library(rainbow)

survey_data <- read.csv("https://arp.numbat.space/week4/survey_data.csv")

survey_data |>
  select(-RespondentID) |>
  group_by(Gender) |>
  count(Satisfaction)
```

https://arp.numbat.space/week4/survey_dplyr_bug.R

Non-interactive debugging

- Necessary for debugging code that runs in a non-interactive environment.
- Is the global environment different? Have you loaded different packages? Are objects left from previous sessions causing differences?
- Is the working directory different?
- Is the `PATH` environment variable, which determines where external commands (like `git`) are found, different?
- Is the `R_LIBS` environment variable, which determines where `library()` looks for packages, different?

Non-interactive debugging

- `dump.frame()` saves state of R session to file.

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

# In a later interactive session ----
load("last.dump.rda")
debugger()
```

- Last resort: `print()`: slow and primitive.

Other tricks

- `sink()` : capture output to file.
- `options(warn = 2)` : turn warnings into errors.
- `rlang::with_abort()` : turn messages into errors.
- If R or RStudio crashes, it is probably a bug in compiled code.
- Post minimal reproducible example to Posit Community or Stack Overflow.

Outline

1 Debugging

2 Styling

3 Profiling

4 Efficiency

Style guides

Tidyverse

<https://style.tidyverse.org/>

Google

<https://google.github.io/styleguide/Rguide.html>

Indentation

- Use **2 spaces** per indentation level.
- Add spaces around operators: `x <- y + z`.

Naming (functions, arguments, objects)

Be brief but descriptive with object names.

Use a consistent naming convention:

- camelCase
- snake_case
- PascalCase

- **Modularity:** Create re-usable parts for maintainability and scalability.
- **Simplicity:** Keep the interface intuitive and easy to use with straightforward interactions.
- **Flexibility:** Allow adaptability to different use cases and user preferences.
- **Feedback:** Provide clear and timely feedback to inform users of actions, errors, and system states.

Automatic styling

- styler: <https://styler.r-lib.org/>
- air: <https://posit-dev.github.io/air/>

These can be configured to automatically style your code when you save.

You can also check your code for common problems with lintr.

Outline

1 Debugging

2 Styling

3 Profiling

4 Efficiency

Profiling functions

- `Rprof()` : records every function call.
- `summaryRprof()` : summarises the results.
- `profvis()` : visualises the results.

Where are the bottlenecks in your code?

```
library(profvis)
library(bench)
f <- function() {
  pause(0.1)
  g()
  h()
}
g <- function() {
  pause(0.1)
  h()
}
h <- function() {
  pause(0.1)
}
```

Profiling

```
tmp <- tempfile()
Rprof(tmp, interval = 0.1)
f()
Rprof(NULL)
writeLines(readLines(tmp))
#> sample.interval=100000
#> "pause" "g" "f"
#> "pause" "h" "g" "f"
#> "pause" "h" "f"
```

Profiling

```
source(here::here("week4/profiling-example.R"))  
profvis(f())
```



Microbenchmarking

system.time()

```
x <- rnorm(1e6)
system.time(min(x))
```

| user | system | elapsed |
|-------|--------|---------|
| 0.002 | 0.000 | 0.002 |

```
system.time(sort(x)[1])
```

| user | system | elapsed |
|-------|--------|---------|
| 0.087 | 0.005 | 0.093 |

```
system.time(x[order(x)[1]])
```

| user | system | elapsed |
|-------|--------|---------|
| 0.086 | 0.000 | 0.086 |

Microbenchmarking

bench::mark()

```
bench::mark(  
  min(x),  
  sort(x)[1],  
  x[order(x)[1]]  
)
```

A tibble: 3 x 6

| | expression | min | median | `itr/sec` | mem_alloc | `gc/sec` |
|---|----------------|----------|----------|-----------|-----------|----------|
| | <bch:expr> | <bch:tm> | <bch:tm> | <dbl> | <bch:byt> | <dbl> |
| 1 | min(x) | 1.17ms | 1.3ms | 752. | 0B | 0 |
| 2 | sort(x)[1] | 64.94ms | 76.3ms | 13.5 | 11.44MB | 10.2 |
| 3 | x[order(x)[1]] | 44.7ms | 45.8ms | 21.2 | 3.81MB | 3.03 |

Microbenchmarking

- `mem_alloc` tells you the memory allocated in the first run.
- `n_gc` tells you the total number of garbage collections over all runs.
- `n_itr` tells you how many times the expression was evaluated.
- Pay attention to the units!

Exercises

2

What's the fastest way to compute a square root?

Compare:

- ▶ `sqrt(x)`
- ▶ `x^0.5`
- ▶ `exp(log(x) / 2)`

Use `system.time()` find the time for each operation.

Repeat using `bench::mark()`. Why are they different?

Outline

1 Debugging

2 Styling

3 Profiling

4 Efficiency

Vectorization

- Vectorization is the process of converting a repeated operation into a vector operation.
- The loops in a vectorized function are implemented in C instead of R.
- Using `map()` or `apply()` is **not** vectorization.
- Matrix operations are vectorized, and usually very fast.

Exercises

Write the following algorithm to estimate $\int_0^1 x^2 dx$ using vectorized code

Monte Carlo Integration

- a. Initialise: `hits = 0`
- b. for `i` in `1:N`
 - ▶ Generate two random numbers, U_1, U_2 , between 0 and 1
 - ▶ If $U_2 < U_1^2$, then `hits = hits + 1`
- c. end for
- d. Area estimate = `hits/N`

Exercises

4 Use `bench::mark()` to compare the speed of `sq()` and `memo_sq()`.