MONASH University

MONASH BUSINESS SCHOOL

ETC4500/ETC5450 Advanced R programming

Week 3: Debugging

Outline

1. Debugging
2. Measuring performance
3. Improving performance
4. Caching

Outline

Outline

Overall debugging strategy

- Google
- Stack Overflow
- Posit Community
- Create a minimal reproducible example
- Create a unit test
- Figure out where the test fails
- Fix it and test

Minimal reproducible examples

- A minimal data set. Use a small built-in dataset, or make a small example.
- If you must include your own data, use `dput()`, but subset where possible.
- The *minimal* amount of code to reproduce the problem. Load only necessary packages.
- If the example involves random numbers, set the seed with `set.seed()`.
- Information about package versions, R version, OS. Use `sessioninfo::session_info()`.

reprex

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 via `reprex(..., session_info = TRUE)`.
- Use the RStudio addin.

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

```r
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

```r
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

```r
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
}
```

```r
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

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

Exercises

1 What's wrong with this code?

```r
# 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`

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

Profiling functions

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

Profiling

Where are the bottlenecks in your code?

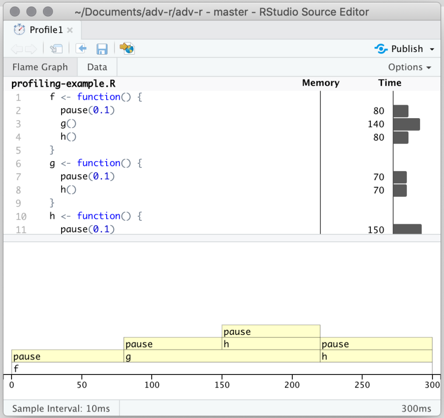
```r
library(profvis)
library(bench)
f <- function() {
  pause(0.1)
  g()
  h()
}
g <- function() {
  pause(0.1)
  h()
}
h <- function() {
  pause(0.1)
}
```

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()

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

```
   user  system elapsed
  0.001   0.000   0.002
```

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

```
   user  system elapsed
  0.062   0.005   0.067
```

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

```
   user  system elapsed
  0.053   0.000   0.054
```

Microbenchmarking

bench::mark()

```r
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)           1.47ms   1.52ms     633.         0B     0
2 sort(x)[1]      84.81ms 103.46ms      10.2   11.44MB     6.77
3 x[order(x)[1]]  52.41ms  53.91ms      18.1    3.81MB     2.26
```

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. Measuring performance
3. Improving performance
4. Caching

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.

Beware of over-vectorising

- Change all missing values in a data frame to zero:

```r
x[is.na(x)] <- 0
```

or

```r
for(i in seq(NCOL(x))) {
  x[is.na(x[, i]), i] <- 0
}
```

Why might the second approach be preferred?

Exercises

3. 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

Outline

1. Debugging
2. Measuring performance
3. Improving performance
4. Caching

Caching: using rds

```r
if (file.exists("results.rds")) {
  res <- readRDS("results.rds")
} else {
  res <- compute_it()  # a time-consuming function
  saveRDS(res, "results.rds")
}
```

Caching: using rds

```r
if (file.exists("results.rds")) {
  res <- readRDS("results.rds")
} else {
  res <- compute_it()  # a time-consuming function
  saveRDS(res, "results.rds")
}
```

Equivalently…

```r
res <- xfun::cache_rds(
  compute_it(), # a time-consuming function
  file = "results.rds"
)
```

Caching: using rds

```r
compute <- function(...) {
    xfun::cache_rds(rnorm(6), file = "results.rds", ...)
}
compute()
```

```
[1]  0.359 -0.257 -0.249 -0.250 -0.156  0.371
```

```r
compute()
```

```
[1]  0.359 -0.257 -0.249 -0.250 -0.156  0.371
```

```r
compute(rerun = TRUE)
```

```
[1] -1.550 -0.479 -0.290  1.103  0.423 -1.650
```

```r
compute()
```

```
[1] -1.550 -0.479 -0.290  1.103  0.423 -1.650
```

Caching: Rmarkdown

````
```{r import-data, cache=TRUE}
d <- read.csv('my-precious.csv')
```


```{r analysis, dependson='import-data', cache=TRUE}
summary(d)
```
````

- Requires explicit dependencies or changes not detected.
- Changes to functions or packages not detected.
- Good practice to frequently clear cache to avoid problems.
- targets is a better solution: Week 8

Caching: Quarto

````
```{r}
#| label: import-data
#| cache: true
d <- read.csv('my-precious.csv')
```


```{r}
#| label: analysis
#| dependson: import-data
#| cache: true
summary(d)
```
````

- Same problems as Rmarkdown
- targets is a better solution: Week 8

Caching: memoise

Caching stores results of computations so they can be reused.

```r
library(memoise)
sq <- function(x) {
  print("Computing square of 'x'")
  x**2
}
memo_sq <- memoise(sq)
memo_sq(2)
```

```
[1] "Computing square of 'x'"
```

```
[1] 4
```

```r
memo_sq(2)
```

```
[1] 4
```

Exercises

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