Computational Skills for Biostatistics I: Lecture 5

Amy Willis, Biostatistics, UW

October 19, 2017

Debugging

"Finding your bug is a process of confirming the many things that you believe are true — until you find one which is not true."
—Norm Matloff

"Defensive programming is the art of making code fail in a well-defined manner even when something unexpected occurs. A key principle of defensive programming is to "fail fast": as soon as something wrong is discovered, signal an error." —Hadley Wickham

Example

```
f <- function(a) g(a)
g <- function(b) h(b)
h <- function(c) i(c)
i <- function(d) "a" + d
f(10)
```

```
## Error in "a" + d: non-numeric argument to binary operato
```

RStudio

```
> f <- function(a) g(a)
> g <- function(b) h(b)
> h <- function(c) "a" * c
> f(10)
```

```
Error in "a" * c : non-numeric argument to binary operator
```

Show Traceback

Rerun with Debug

```
>
```

Showing `traceback`

```
> f <- function(a) g(a)
> g <- function(b) h(b)
> h <- function(c) "a" * c
> f(10)
```

```
Error in "a" * c : non-numeric argument to binary operator
```

Hide Traceback

Rerun with Debug

```
 3. h(b)
 2. g(a)
 1. f(10)
```

Condition handling

```
f1 <- function(x) {
  x^2
  0
}
f1("x")
```

```
## Error in x^2: non-numeric argument to binary operator
```

try()

```
f1 <- function(x) {
  try(x^2)
  0
}
f1("x")
```

```
## [1] 0
```

An error is printed, but execution continues!

try()

Longer sections of code can be wrapped:

```
try({
  a <- 10
  b <- "a"
  a + b
})
```

Condition handling

```
success <- try(1 + 2)
class(success)
```

```
## [1] "numeric"
```

```
failure <- try("a" + 2)
class(failure)
```

```
## [1] "try-error"
```

Ensure some output is produced

```
std_error <- Inf
try(std_error <- my_risky_std_error("bad-input.csv"),
    silent = TRUE)
std_error
```

```
## [1] Inf
```

Creating an error

```
square_variable <- function(x) {
  if (class(x) == "numeric") {
    x^2
  } else {
    stop("Non-numeric argument:
         I can only square numbers!\n")
  }
}
square_variable(5)
```

```
## [1] 25
```

```
square_variable("5")
```

```
## Error in square_variable("5"): Non-numeric argument:
##          I can only square numbers!
```

Creating an error

```
square_variable <- function(x) {
  stopifnot(class(x) == "numeric")
  x^2
}
square_variable(5)
```

```
## [1] 25
```

```
square_variable("5")
```

```
## Error: class(x) == "numeric" is not TRUE
```

Creating a warning

```
square_root <- function(x) {
  if (x >= 0) {
    sqrt(x)
  } else {
    warning("Negative argument:
            Complex numbers produced!\n")
    sqrt(x + 0i)
  }
}
square_root(5)
```

```
## [1] 2.236068
```

```
square_root(-5)
```

```
## Warning in square_root(-5): Negative argument:
##             Complex numbers produced!

## [1] 0+2.236068i
```

Creating a message

```
sample_normals <- function(x) {
  if (x %% 1 == 0) {
    my_sample <- rnorm(x)
  } else {
    message("Input not an integer; rounding")
    my_sample <- rnorm(round(x))
  }
  my_sample
}
sample_normals(2.01)
```

```
## Input not an integer; rounding
```

```
## [1] -0.6011451  0.3285288
```

Messages > printing output

```
chatty_function <- function() {
  paste("Hi, Biostat561!")
}
chatty_function()
```

```
## [1] "Hi, Biostat561!"
```

```
silenceable <- function() {
  message("Hi, Biostat561!")
}
silenceable()
```

```
## Hi, Biostat561!
```

```
suppressMessages(silenceable())
```

Local and global suppression

`suppressMessages()`, `suppressWarnings()` can wrap noisy functions; use `options(warn = -1)` to silence globally

```
options(warn=-1)
warning("Reviewers can be mean!")
options(warn=0)
warning("Seattle has earthquake risk!")
```

```
## Warning: Seattle has earthquake risk!
```

Best practice

Be very careful when modifying `options()` calls in functions

```
non_invasive_function <- function() {
  oldw <- getOption("warn")
  options(warn = -1)
  # other_function
  options(warn = oldw)
}
```

Huh?

```
silenceable() %>% suppressMessages
```

```
## Hi, Biostat561!
```

```
## NULL
```

Closing out neatly

```
make_plots <- function(x, filename) {
  destination <- paste(filename, ".pdf", sep="")
  pdf(destination)
  hist(rnorm(10))
  hist(x)
  dev.off()
}
make_plots("5", "my_plots")
```

```
## Error in hist.default(x): 'x' must be numeric
```

Closing out neatly with tryCatch

```
plotting_wrapper <- function(x, filename) {
  tryCatch(make_plots(x, filename),
  error = function(c) print("Error in make_plots"),
  warning = function(c) print("Warning in make_plots"),
  finally = {
    if (dev.cur() != 1) {
      print("Closing plotting window")
      dev.off()
    }
  }
  )
}
```

Closing out neatly with tryCatch

```
plotting_wrapper(5, "my_plots")
```

```
## [1] "Closing plotting window"
```

```
## pdf
##   2
```

```
plotting_wrapper("5", "my_plots")
```

```
## [1] "Error in make_plots"
## [1] "Closing plotting window"
```

Always code well, but consider context

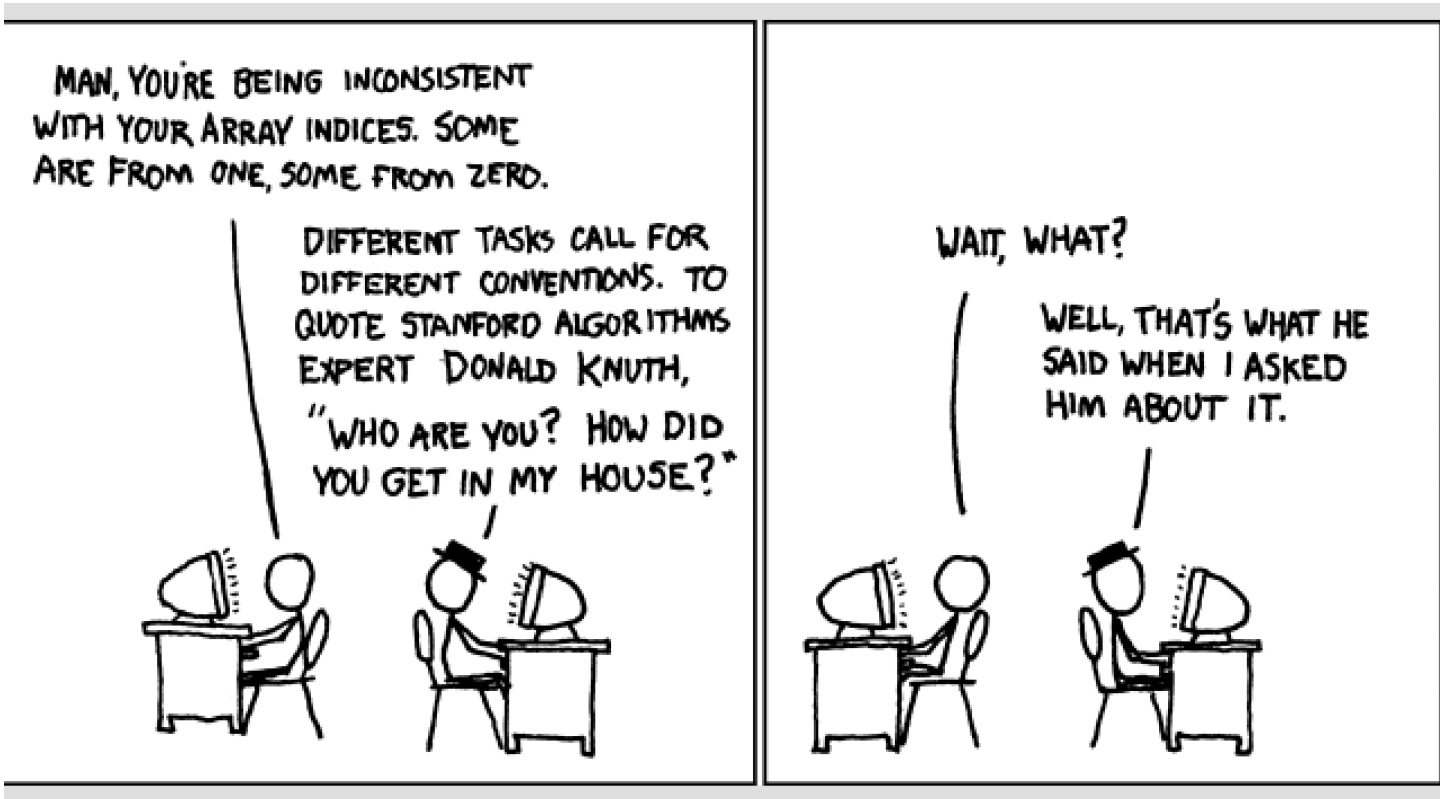
"If you're writing functions to facilitate interactive data analysis, feel free to guess what the analyst wants and recover from minor misspecifications automatically. If you're writing functions for programming, be strict. Never try to guess what the caller wants."
— Hadley Wickham

Profiling code

"We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%. A good programmer will not be lulled into complacency by such reasoning, [she] will be wise to look carefully at the critical code; but only after that code has been identified." —Donald Knuth.

Lots of tools exist for profiling both memory and speed of your code (e.g. `lineprof`, `proftools`, `profr`); use them when you need them

Donald Knuth



Parallelisation

```
library(parallel)
cores <- detectCores()
cores
```

```
## [1] 8
```

```
pause <- function(i) {
  function(x) Sys.sleep(i)
}
system.time(lapply(1:5, pause(0.25)))
```

```
##    user  system elapsed
##   0.000   0.000   1.264
```

```
system.time(mclapply(1:5, pause(0.25), mc.cores = cores))
```

```
##    user  system elapsed
##   0.003   0.008   0.262
```

Coming up

- Next week: LaTex and Markdown by Kelsey and David
- No homework this week!
  - Homework on debugging and defensive programming due in 2 weeks... written in LaTex/Markdown!