Debugging R code

Robert Stojnic <rs550@cam.ac.uk>
Laurent Gatto <lg390@cam.ac.uk>

Cambridge System Biology Centre
University of Cambridge

May 10, 2013

Using R's tools

- Call `traceback()` after error to print the sequence of calls that lead to the error.
- Use `debug(faultyFunction)` to register `faultyFunction` for debugging, so that `browser()` will be called on entry. In `browser` mode, the execution of an expression is interrupted and it is possible to inspect the environment (with `ls()`). Use `undebug(faultyFunction)` to revert to normal usage.
  See exercise on next slide.
- Use `trace()` to insert code into functions, start the `browser` or `recover()` from error.
- Set `options(error=recover)` to get the call stack and browse in any of the function calls.

Good reference: *An Introduction to the Interactive Debugging Tools in R*[a]

[a] `http://www.biostat.jhsph.edu/~rpeng/docs/R-debug-tools.pdf`

Debugging example (1)

Let's walk through an example[1]. The buggy setp is:

```
e <- function(i) {
  x <- 1:4
  if (i < 5) x[1:2]
  else x[-1:2] # oops! x[-(1:2)]
}
f <- function() sapply(1:10, e)
g <- function() f()
```

[1] credit Martin Morgan and Robert Gentleman

Debugging example (2)

```
> g()
Error in x[-1:2] (from #3) : only 0's may be mixed with negati
> traceback()
5: FUN(1:10[[5L]], ...)
4: lapply(X = X, FUN = FUN, ...)
3: sapply(1:10, e) at #1
2: f() at #1
1: g()
```

Debugging example (3)

Using `options(error=recover)`, we are given a list of frames to debug. Once inside a frame, one can view and modify variables.

```
> g()
Error in x[-1:2] (from #3) : only 0's may be mixed with negati

Enter a frame number, or 0 to exit

1: g()
2: #1: f()
3: #1: sapply(1:10, e)
4: lapply(X = X, FUN = FUN, ...)
5: FUN(1:10[[5]], ...)

Selection:
```

Debugging example (4)

```
Selection: 5
[...]
Browse[1]> ls()
[1] "i" "x"
Browse[1]> c

Enter a frame number, or 0 to exit

1: g()
2: #1: f()
3: #1: sapply(1:10, e)
4: lapply(X = X, FUN = FUN, ...)
5: FUN(1:10[[5]], ...)

Selection: 0
options(error = NULL)
```

Using the debugger directly

```
> debug(e)
> g()
debugging in: FUN(1:10[[1L]], ...)
debug at #1: {
    x <- 1:4
    if (i < 5)
        x[1:2]
    else x[-1:2]
}
Browse[2]> debug at #2: x <- 1:4
Browse[2]> Q
> undebug(e)
```

Using trace

```
## Report whenever e invoked
trace(e)
## Evaluate arbitrary code whenever e invoked
trace(e, quote(cat("i am", i, "\n")))
## Another way to enter browser whenver e invoked
trace(e, browser)
## stop tracing
untrace(e)

require("stats4")
## Debug S4 methods with signature argument
trace("plot", browser, exit = browser, signature = c("track",
```

Calling with custom handlers (1)

The `withCallingHandlers` function allows to defined special behaviour in case of /unusual conditions/, including warnings and errors. In the example below, we start a browser in case of (obscure) warnings.

```
> f <- function(x){
+   for(i in 1:10){
+     ## make an example 2x2 contingency table
+     d = matrix(sample(4:10, 4), nrow=2, ncol=2)
+     ## will produce warning if there is a 5 or less
+     ## in the contingency table
+     chisq.test(d)
+   }
+ }
```

Calling with custom handlers (2)

```
> set.seed(1)
> f()
> set.seed(1)
> withCallingHandlers(f(), warning=function(e) recover())
```

Exercise

The `readFasta2` function is similar to `readFasta`, but reads multiple sequences in a fasta file and returns a list of `DnaSeq` instances... at least, that's what it is supposed to do.

1. source the `readFasta2.R` file, and try the function with the `multiple.fasta` file.
2. Prepare for debugging: `debug(readFasta2)`.
3. Debug!
4. `fix` the function.

Hint: when debugging, use `n` (or an empty line) to advance to the next step, `c` to continue to the end of the current context (to the end of a loop for instance), `where` to print the stack trace of all active function calls and `Q` to exit the `browser`.
Other hint: use `ls(all.names=TRUE)` to see all objects, also those that start with a '`.`'.