Lecture 13: Waiter, There's a Bug In My Code

36-350

8 October 2014

How Do We Fix Errors?

We find errors in code all the time:

- Minor: Unexpected results

```
-0.5^0.5
```

```
## [1] -0.7071
```

```
(-0.5)^0.5
```

```
## [1] NaN
```

How Do We Fix Errors?

We find errors in code all the time:

- Intermediate: Doesn't work, but you know it

```
mean(NA)
```

```
## [1] NA
```

How Do We Fix Errors?

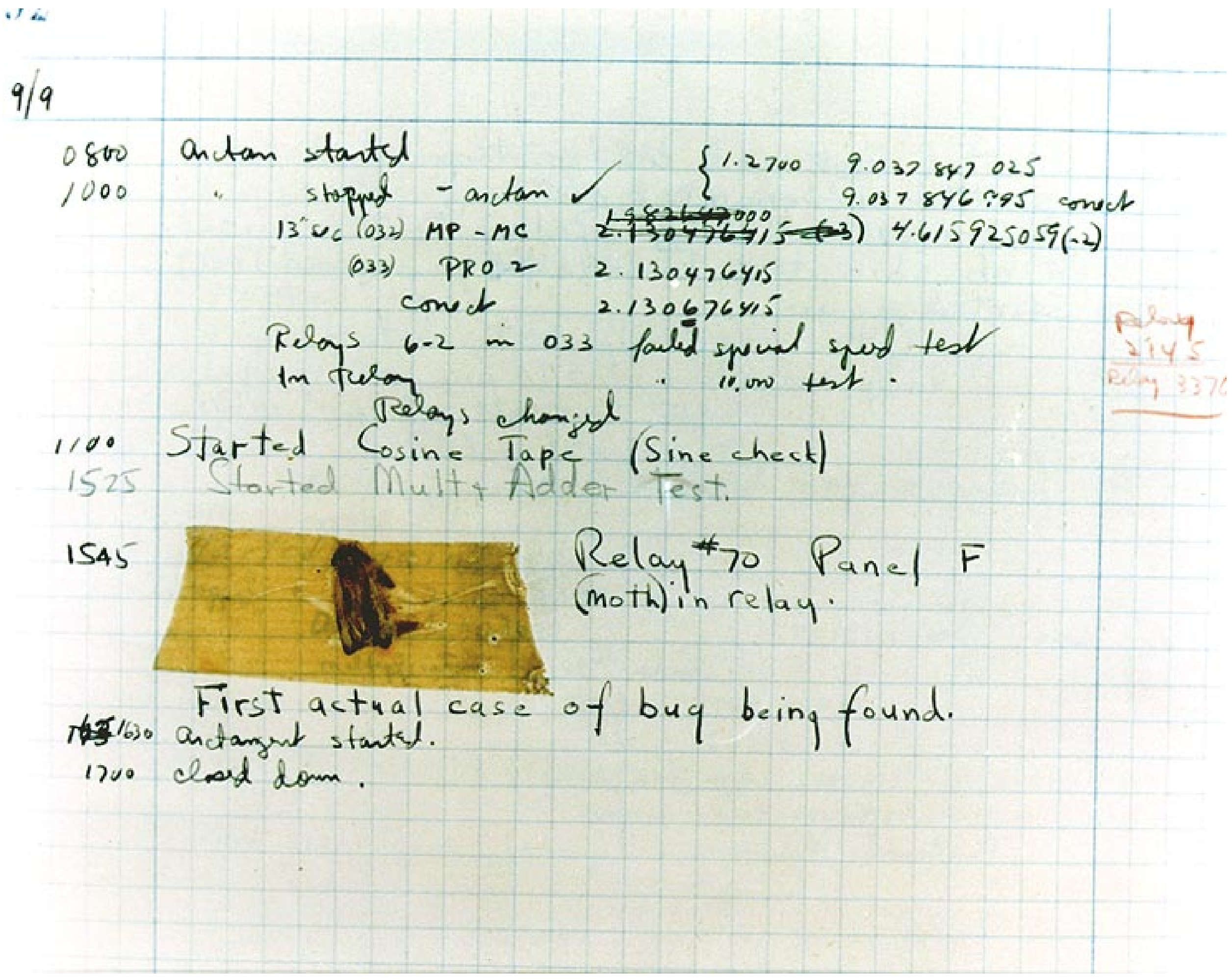
We find errors in code all the time:

- Major: Doesn't Work, But It's Not Obvious

Bug!

The original name for glitches and unexpected defects: dates back to at least 1876 (Edison), but better story from Grace Hopper, PhD, 1946:

Bug!



9/9

0800 Arctan started
1000 " stopped - arctan ✓ {1.2700 9.037 847 025
9.037 846 995 conect
13" uc (032) MP - MC ~~1.982147000~~ ~~2.130476415 (-3)~~ 4.615925059(-2)
(033) PRO 2 2.130476415
conect 2.130676415
Relays 6-2 in 033 failed special speed test
In relay " 10.000 test.
Relays changed
1100 Started Cosine Tape (Sine check)
1525 Started Mult + Adder Test.
1545 Relay #70 Panel F
(moth) in relay.
First actual case of bug being found.
~~[illegible]~~ 1630 Arctangent started.
1700 closed down.

Relay 2145
Relay 3370

Stages of Debugging

Debugging is (largely) Differential Diagnosis:

1. Characterize the error: what exactly is going wrong?
2. Localize the error: where in the code does the mistake originate?
3. Modify the code: did you eliminate the error? Did you add new ones?

Characterize the Bug

First step: Reproduce the Error

- Can we produce it repeatedly when re-running the same code, with the same input values?
- In particular: If we start the same code in a clean copy of R (say, on another machine), does the same thing happen?

Characterize the Bug

Second step: Bound the error

- How much can we change the inputs and get the same error?
- A different error?
- How big is the error?

Characterize the Bug

Third step: Get more information

- Add extra output to the function
- Add intermediate output using `message()` or `print()`

Localizing The Bug

Worst worst case: the problem is a diffuse, all-pervading wrongness and you should curse the names of your professors and the patron saints of computing (dang it, Tukey!)

Typical case: the problem is a far more localized issue or set of issues that can be pinned down.

Localizing The Bug

Tools: - `traceback()`: where did an error message come from? - `message(), print()`: present intermediate outputs. - `warning()`: message from the code when it's finished running - `stop(), stopifnot()`: terminate the run if something's wrong - Controlled inputs - Interactive debugging tools

Common Issues: Syntax

- Parenthesis mis-matches
- `[[ ... ]]` vs. `[ ... ]`
- `==` vs. `=`
- Identity of floating-point numbers
- Vectors vs. single values: code works for one value but not multiple ones, unexpected recycling
- Element-wise comparison of structures (use `identical`, `all.equal`)
- Silent type conversions

Common Issues: Logic

- Confusing variable names
  - Try to avoid single-letter names, insists Prof. Thomas
- Confusing function names
- Giving unnamed arguments in the wrong order
- R expression does not match the math you mean (left something out, added something)

Common Issues: Scope and Global Variables

- Relying on a global variable which doesn't have the right value — or only has the right value in *one* situation)
- Assuming that changing a variable inside the function will change it elsewhere
- Confusing variables within a function and those from where the functional was called

Simple Functional Test

Recall the Gamma distribution estimator from a previous lab session. What if this was the code:

```r
gamma.est <- function(input) {
  meaninput <- mean(input)
  varinput <- var(input)
  scale <- varinput/meaninput
  shape <- meaninput/scale
  output <- list(shape=shape,scale=scale)
  return(output)
}
```

Simple Functional Test

First, make sure it works! Test this function on its own. A simple unit test:

```r
input <- rgamma(10000, shape=0.1, scale=10)
gamma.est(input)
```

```
## $shape
## [1] 0.103
##
## $scale
## [1] 9.646
```

traceback()

Literally: traces back through all the function calls leading to the last error.

Start your attention at the first of these functions which you wrote (not that it can't be someone else at fault).

Often the most useful bit is somewhere in the middle (there may be many low-level functions called, like `mean()`)

`traceback()` Example: Jackknife

Now, the jackknife estimator for the parameter standard error:

```
#datavector = cats$Hwt[1:3]
gamma.jackknife <- function(datavector) {
  datalength <- length(datavector)
  jack.estimates <- sapply(1:length(datavector),
    function (omitted.point) unlist(gamma.est(datavector[-omitted.point])))
  var.of.ests <- apply(jack.estimates,1,var)
  jack.var <- ((datalength-1)^2/datalength)*var.of.ests
  return(sqrt(jack.var))
}
```

traceback() Example: Jackknife

What happens?

```
library(MASS)
```

```
> gamma.jackknife(cats$Hwt[1:3])
 Error: is.atomic(x) is not TRUE
> traceback()

5: stop(sprintf(ngettext(length(r), "%s is not TRUE", "%s are not all TRUE"),
       ch), call. = FALSE, domain = NA)
4: stopifnot(is.atomic(x))
3: FUN(newX[, i], ...)
2: apply(jack.estimates, 1, var) at #4
1: gamma.jackknife(cats$Hwt[1:3])
```

Add intermediate messages

`print()` forces values to the screen; stick it before the problematic part to see if values look funny

```
print(paste("x is now",x))
y <- a.tricky.function(x)
print(paste("y has become",y"))
```

Add intermediate messages

Add `print(str(jack.estimates))` before the `apply()` and run again:

```
gamma.jackknife.2 <- function(datavector) {
  datalength <- length(datavector)
  jack.estimates <- sapply(1:length(datavector),
    function (omitted.point) gamma.est(datavector[-omitted.point]))
  print(str(jack.estimates))
  var.of.ests <- apply(jack.estimates,1,var)
  jack.var <- ((datalength-1)^2/datalength)*var.of.ests
  return(sqrt(jack.var))
}
```

Add intermediate messages

```
> gamma.jackknife.2(cats$Hwt[1:3])

List of 6
 $ : num 32.4
 $ : num 0.261
 $ : num 21.8
 $ : num 0.379
 $ : num 648
 $ : num 0.0111
 - attr(*, "dim")= int [1:2] 2 3
 - attr(*, "dimnames")=List of 2
  ..$ : chr [1:2] "shape" "scale"
  ..$ : NULL
NULL
 Show Traceback

 Rerun with Debug
 Error: is.atomic(x) is not TRUE
```

What happened?

The problem is that `gamma.est` gives a list, and so we get a weird list structure, instead of a plain array

Solution: re-write `gamma.est` to give a vector (as in the lab instructions), or wrap `unlist` around its output

```r
gamma.est <- function(input) {
  meaninput <- mean(input)
  varinput <- var(input)
  scale <- varinput/meaninput
  shape <- meaninput/scale
  return(c(shape=shape,scale=scale))
}
```

warning()

Print warning messages along with the call that initiated the weirdness

```r
quadratic.solver <- function(a,b,c) {
  determinant <- b^2 - 4*a*c
  if (determinant < 0) {
    warning("Equation has complex roots")
    determinant <- as.complex(determinant)
  }
  return(c((-b+sqrt(determinant))/2*a, (-b-sqrt(determinant))/2*a))
}
```

warning()

```
quadratic.solver(1,0,-1)
```

```
## [1]  1 -1
```

```
quadratic.solver(1,0,1)
```

```
## Warning: Equation has complex roots
```

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

stopifnot()

Halt when results aren't as we expect, and say why. Recall: we've seen this one before

N.B., once you have found the bug, it's generally good to turn lots of these off!

Test Cases and Dummy Functions

Localize error by using inputs where you know the answer.

If you suspect `myfunction()` is buggy, give it a simple case where the proper output is easy for you to calculate "by hand" (i.e., not using your implementation)

If `myfunction()` works on a bunch of cases, well and good; if not, you need to fix it (and possibly other things).

If inputs come from other functions, write functions, with the right names, to generated fixed, simple values of the right format and content (save the real functions somewhere else)

To make sure the dummy is working, make its output as simple as you can

Interactive Debugging

The `browser`, `recover` and `debug` functions modify how R executes other functions

Let you view and modify the environment of the target function, and step through it

You do *not* need to master them for this class, though they can be very helpful

See chapter 13 of Matloff, and §§ 3.5–3.6 of Chambers

Making a Change

After diagnosis, treatment: once the error is characterized and localized, guess at what's wrong with the code and how to fix it

Try the fix: does it work? Have you broken something else?

Try small cases first!

Programming for Debugging

- The truth of it: you are going to have to debug. You already ought to have practice!
- Debugging is frustrating and time-consuming, but essential.
- Writing now to make it easier to debug later is worth it, even if it takes a bit more time
  - lots of the design ideas in the class contribute to this

Writing for Debugging

- Comment your code!
  - Insist on the three comment lines for each function: purpose, inputs, outputs
  - Comment the innards as well, especially anything which strikes you as tricky or clever
  - If you borrowed an idea from somewhere, use the comment to remind yourself of where (and acknowledge the borrowing)
- Use meaningful names!
  - There are no restrictions on name lengths, and few on name content
  - Avoid abbreviations, unless very well-established conventions (and put in comments explaining the convention)

Designing for Debugging

- Use top-down design and write modular, functional programs
- Respect the interfaces
- Don't write the same code multiple times
- Use tests

Top-Down Programming

- Easier to identify errors, because the job of each function is small and well-characterized
- Easier to localize errors:
  - If a bottom-level function is working, the error must be somewhere up the chain
  - If a function can integrate artificial inputs, the problem has to be either in the inputs its called with, or in a sub-function
- Strategy: get the lowest-level functions right, and then work back up the chain

Interfaces

- Respecting the interface means giving everything needed as part of the input (or context of definition) and only relying on the explicit return value
  - Makes it easier to reproduce bugs
  - Makes it easier to characterize bugs by finding the bad inputs
  - Global variables considered *especially* harmful
  - Special considerations for stochastic simulations, like `set.seed()`

Unified Code

- We often have to do basically similar tasks at multiple points in the program.
- Solutions: either write parallel code for each instance, or a single function called multiple times
- Writing one function is better for debugging:
  - If it's wrong, the error gets propagated everywhere
  - *but* there is only one place that needs fixing
  - *and* there is no chance to introduce new errors by mistakes in copying or adjustment

Tests

Help answer "How do I know I've fixed this bug?"

Help answer "How do I know I haven't broken something that was working?"

Much of what you did to characterize and localize the bug can be turned into tests

Error Handling

Ordinarily, errors just lead to crashing or the like

R has an **error handling** system which allows your function to catch, and recover from, errors in functions they call (functions: `try`, `tryCatch`)

Can also recover from not-really-errors (like optimizations that don't converge)

This system is very flexible, but rather complicated

Summary

- Debugging is largely about differential diagnosis
- When you find a bug, characterize it by making sure you can reproduce it, and figure out what inputs do and don't give the error
- Once you know what the bug does, localize it by traceback and adding messaging from the code; by dummy input generators; and by interactive tracing
- Examine the localized error for syntax error and for logical errors; fix them, and see if that gets rid of the bug without introducing new ones
- Program for debugging: write with comments and meaningful names; write modular functions; avoid repeated code

Next time: more about testing