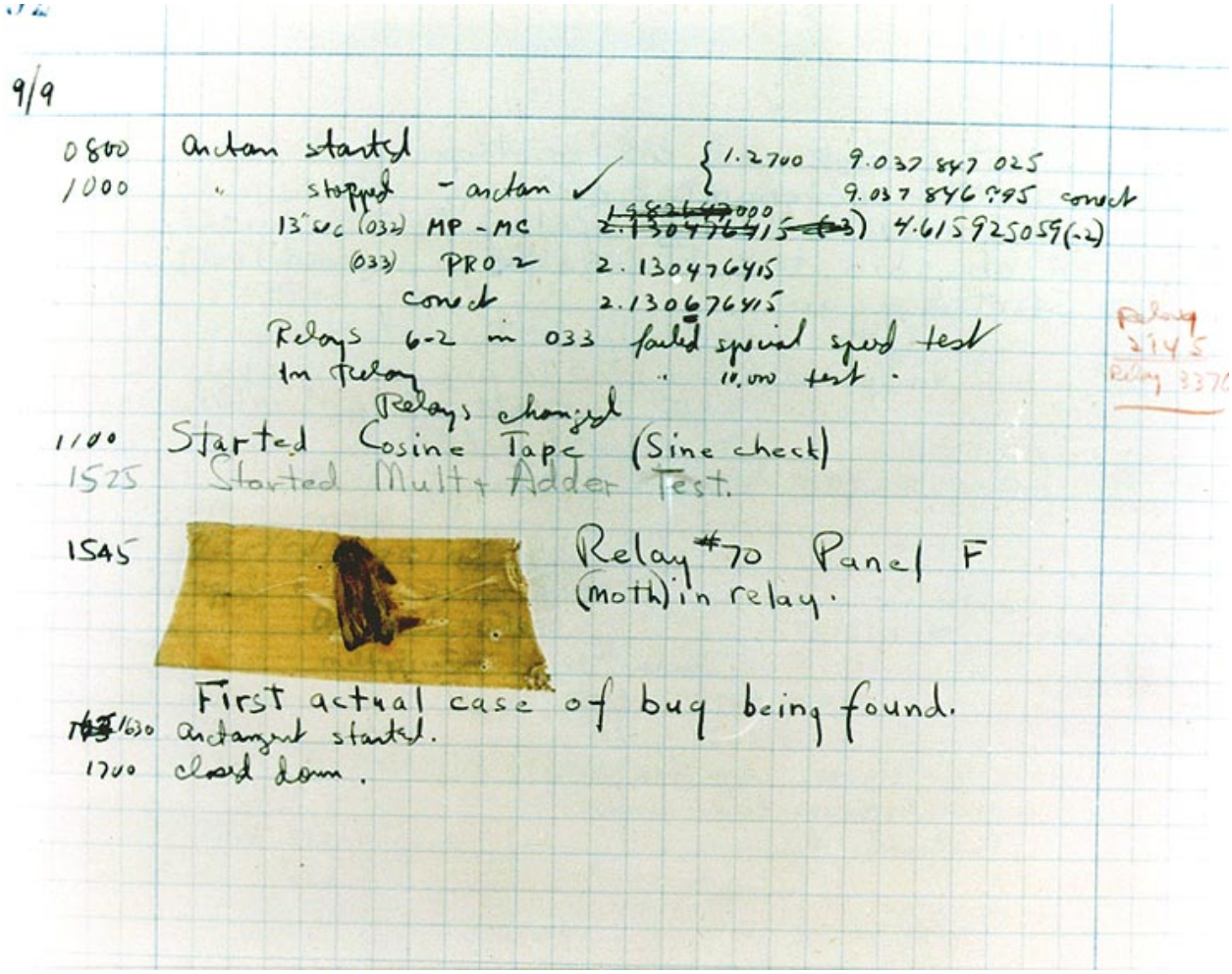
Lecture 10: Waiter, There is a Bug in My Code!

Statistical Computing, 36-350

Wednesday October 14, 2015

Bug!

The original name for glitches and unexpected defects: dates back to at least Edison in 1876, but better story from Grace Hopper in 1947:



Stages of debugging

Debugging is (largely) a process of differential diagnosis:

0. Reproduce the error: can you make the bug re-appear?
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?

Reproduce the bug

Step 0: make if happen again

- 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

Step 1: figure out if it's a pervasive/big problem

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

Localizing the bug

Step 2: find out where things are going wrong

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

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
- 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 function was called

Example: Gamma estimation

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

```
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 test

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

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

```
## $shape
## [1] 0.09764142
##
## $scale
## [1] 10.26658
```

`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()`)

Example: jackknife

Now, the jackknife estimator for the parameter standard error:

```
gamma.jackknife = function(dat) {
  n = length(dat)
  jack.estimates = sapply(1:n, function(i){
      return(gamma.est(dat[-i]))})
  sd.of.ests = apply(jack.estimates,1,sd)
  return(sd.of.ests*sqrt((n-1)^2/n))
}
```

What happens?

```
library(MASS)
#gamma.jackknife(cats$Hwt)
```

If you uncommented the above line, you would have seen this:

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

Then, if we called `traceback()`:

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

Adding intermediate messages

`print()` forces values to the screen; stick it in a bunch of places to see if things are you expect

```
gamma.jackknife.2 = function(dat) {
  print("Spot #1"); print(head(dat))
  n = length(dat)
  jack.estimates = sapply(1:n, function(i){
      return(gamma.est(dat[-i]))})
  print("Spot #2"); print(jack.estimates[,1])
  sd.of.ests = apply(jack.estimates,1,sd)
  print("Spot #3"); print(sd.of.ests,"\n")
  return(sd.of.ests*sqrt((n-1)^2/n))
}
```

```
#gamma.jackknife.2(cats$Hwt)
```

If you uncommented the above line, you would have seen this:

```
> gamma.jackknife.2(cats$Hwt)
[1] "Spot #1"
[1] 7.0 7.4 9.5 7.2 7.3 7.6
[1] "Spot #2"
$shape
[1] 19.32514

$scale
[1] 0.5514032

 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, or alternatively, wrap `unlist` around its output

```
gamma.est.2 = function(input) {
  meaninput = mean(input)
  varinput = var(input)
  scale = varinput/meaninput
  shape = meaninput/scale
  return(c(shape,scale))
}

gamma.jackknife.3 = function(dat) {
  n = length(dat)
  jack.estimates = sapply(1:n, function(i){
      return(gamma.est.2(dat[-i]))})
  sd.of.ests = apply(jack.estimates,1,sd)
  return(sd.of.ests*sqrt((n-1)^2/n))
}

gamma.jackknife.3(cats$Hwt)
```

```
## [1] 2.74062279 0.07829436
```

warning()

`warning()` allows you to print warnings to the console, can be helpful to others who might be using your code, and could encounter errors

```
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))
}

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

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

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

```
## Warning in quadratic.solver(1, 0, 1): Equation has complex roots
```

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

`stop()` and `stopifnot()`

These halt the function execution when we see something not expected. Could prevent bugs downstream, if users pass arguments that aren't as we expect

We've seen examples of these before; they're smart to put in your code

More advice

- Start small: if you find encounter a bug in a long piece of code, break it down, and start chunking up small bits and running them, to see if you can figure out what's wrong
- Step through the function: it's not fancy, but sometimes it's actually helpful to just step through each line of a function manually. Note: for this, you're going to have to define all of its arguments appropriately
- 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 don't need to know them for this class, but they can be very helpful

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. See `try()`, `tryCatch()`

```
tryCatch({
  gamma.sds = gamma.jackknife(cats$Hwt)
}, error = function(err) {
  gamma.sds = NA
  cat(paste("Encountered the following error:\n",
              err$message,
              "\nContinuing on without jackknife estimates")) })
```

```
## Encountered the following error:
##  is.atomic(x) is not TRUE
## Continuing on without jackknife estimates
```

You don't need to know these for this class, but just note that they can be critically important, so that your code doesn't completely crash an burn from small errors!

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
- Use lots of comments, use meaningful variable names!

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