561

Computational Skills for Biostatistics I

Ken Rice

UW Dept of Biostatistics

October 11, 2016

Previously...

- Objects and Classes
- Generic functions
- A little about debugging

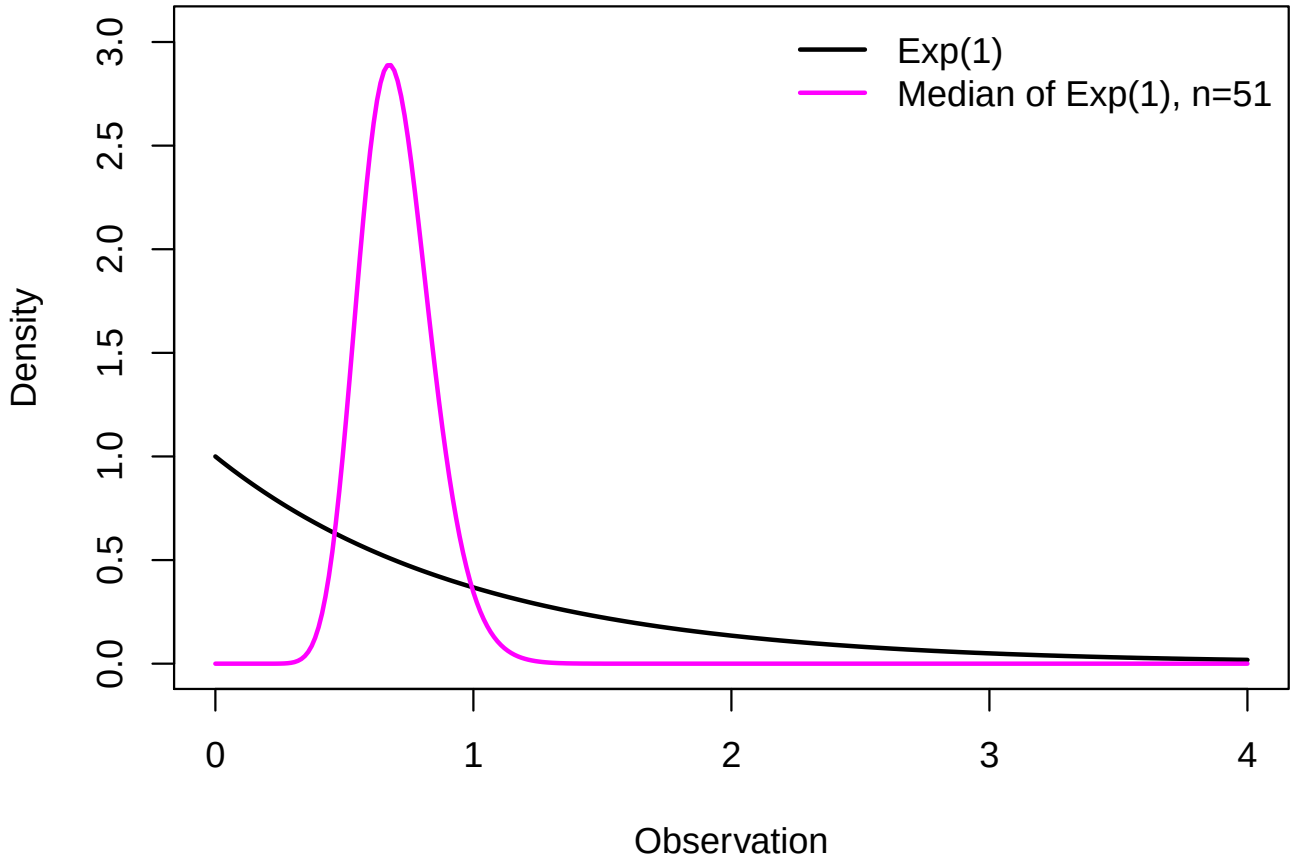
Coming up;

- Writing your own functions
- Applying them to objects – repeatedly
- Debugging your function and the loop it's in

Replicating commands

A question from analysis of survival traits – and its answer!

What is the expected value of the median of a sample, size $n = 51$, of independent observations from $Exp(1)$? What is the variance of the sample median?



Replicating commands

The picture didn't make it obvious? Here are the **exact** answers;

$$\mathbb{E}[\,\text{Median}_{51}\,] = \frac{217817893653910867 4153}{309904450424599670 6400}$$

$$\mathbb{E}[\,\text{Median}^2_{51}] = \frac{24672823160636679674592332321392579768019 59}{48020384196486577490012788153798239004800 00}$$

These are 0.70286 and 0.51380 to 5 d.p. – so the variance is 0.51380 - 0.702862^2= 0.01978 (Standard deviation is ≈0.14)

- Yes, there are 'pretty' answers here
- In general there aren't – but the 'expectation' $\mathbb{E}[\ldots]$ just means averaging over lots of datasets – which computers are good at.

Replicating commands

To get a computer to do this job, we'll write code that;

1. Generates a sample of size $n = 51$ from $Exp(1)$
2. Calculates its median, and returns this number
3. Replicates steps 1 and 2 many times, then works out the mean and variance of the stored numbers

Some code doing this, that might be familiar to you;

```
bigB <- 10000
many.medians <- rep(NA, bigB)
set.seed(4)
for(i in 1:bigB){
   mysample <- rexp(n=51, rate=1) # take a sample, size 51
   many.medians[i] <- median(mysample)
   }
mean(many.medians)
var(many.medians)
```

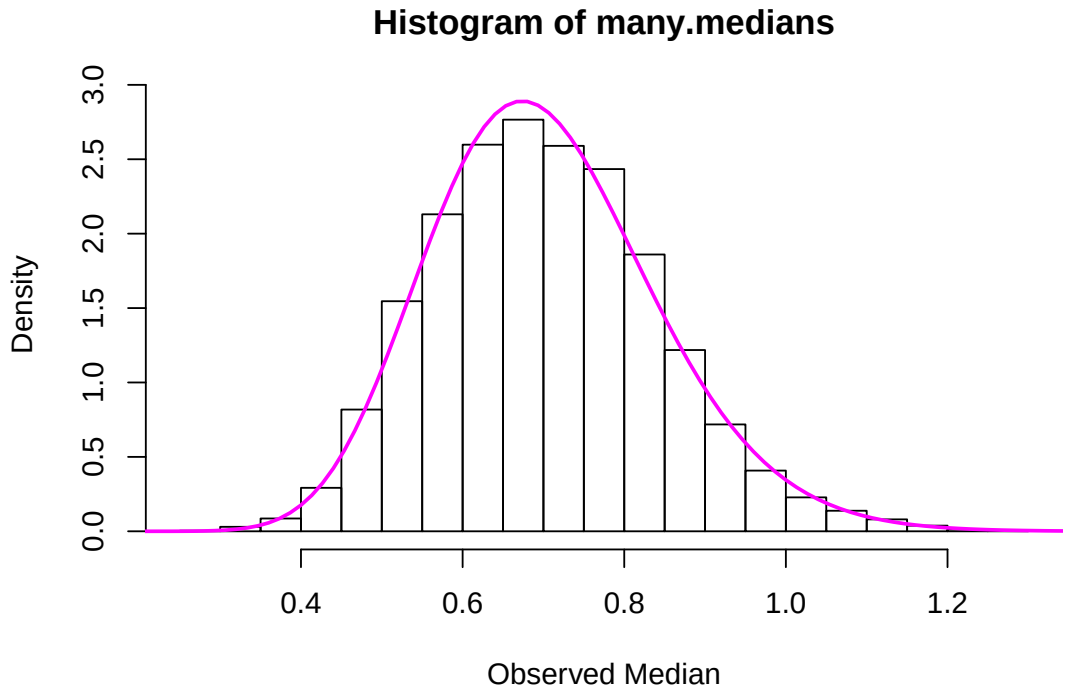
The iteration is done with a `for()` loop.

Replicating commands

What were the answers?

```
> mean(many.medians)
[1] 0.702171          # exact answer is 0.70286
> var(many.medians)
[1] 0.01955728        # exact answer is 0.01978
```

NB: for large-enough values of 10000, we could work basically *anything* about the sample median, with little extra work;



Replicating commands

Mental model for `set.seed()`;

Setting the ‘seed’ makes your work reproducible. Use any value.

Replicating commands

The `for()` loop is not terrible – we will use it throughout 514 – but do watch out;

- First make an empty object (e.g. vector, matrix, data frame) of the right dimensions and **then** fill it in
- For large loops/objects, 'growing' the output in R is a **big** slowdown – because of the way memory is handled
- Do remember to store output – and in the right place, at the right point in the *expression* between the curly brackets
- The index `i` – also known as a *counter* – is pointless here; it's not used except for storage, i.e. admin

Note: we looped over `i` in vector `1:bigB`, but you can loop over any vector – a vector of IDs, hair colors, etc etc.

Replicating commands

R can implement loops with *functional programming* – we just tell R what code we want run many times over, and let it take care all of the admin.

The simplest way to do these is with `replicate()`

```
set.seed(4)
many.medians <- replicate(bigB, {
   mysample <- rexp(n=51, rate=1)
   median(mysample)
   }
)
```

or just

```
set.seed(4)
many.medians <- replicate(bigB, median(rexp(n=51, rate=1)) )
```

- The last object evaluated in the expression is returned – or use the `return()` function for more complex jobs
- One-command expressions don't need curly brackets – but use them anyway, to remind yourself what the code does

Broken down `by()` age and sex

As seen in e.g. 514, Table 1 (or Table 2) in a medical paper often provides means and standard deviations, percentages, or frequency tables of many variables broken down by groups – such as case/control status, age, sex, exposure, etc.

Translating this to R, we need to apply a simple computation to subsets of the data, which are defined `by()` some variable.

A command to do just this is `by()` – see next page for an example. The function `tapply()` is extremely similar, but its name is harder to remember.

Broken down `by()` age and sex

Using the built-in `airquality` dataset;

```
> by(airquality$Ozone, list(month=airquality$Month), mean, na.rm=TRUE)
month: 5
[1] 23.61538
------------------------------------------------------------
month: 6
[1] 29.44444
------------------------------------------------------------
month: 7
[1] 59.11538
------------------------------------------------------------
month: 8
[1] 59.96154
------------------------------------------------------------
month: 9
[1] 31.44828
```

- Literally: using the Ozone data, break into subsets according to Month and calculate the mean of each (omitting NAs)
- In plainer language: calculate month-specific means of Ozone (omitting NAs)

Broken down `by()` age and sex

- The first argument (`data`) is the variable to be analyzed.
- The second argument (`INDICES`) is a list of variable-defining subsets. In this case, a single variable, but we could do `list(month=airquality$Month, toohot=airquality$Temp>85)` to get a breakdown by month and temperature
- The third argument (`FUN`) is the analysis function to use on each subset
- Any other arguments (`na.rm=TRUE`, here) are passed on to the analysis function
- The result is an object of class `by` – which has its own print method, `print.by()` – that produces all the labels, separators etc. You might find this helpful on screen, but not when producing a nicely-formatted table for your paper (or HW)

Broken down `by()` age and sex

To get rid of the formatting, recall `unclass()` from last week,

```
> a <- by(airquality$Ozone, list(month=airquality$Month), mean, na.rm=TRUE)
> unclass(a)
month
       5        6        7        8        9
23.61538 29.44444 59.11538 59.96154 31.44828
attr(,"call")
by.default(data = airquality$Ozone, INDICES = list(month = airquality$Month),
    FUN = mean, na.rm = TRUE)
> unclass(a)[3] # or use unclass(a)["7"], to pick by name
       7
59.11538
```

- Also see the optional `simplify` argument in `by()` (and also in the `tapply()` function) – if `FALSE`, a list is returned; if `TRUE` it will try to provide a vector or array, depending on what `FUN` returns for each subset.
- Yes, we could write a for loop, with `i=5:9`. But using `by()`'s functional approach, there's no explicit subsetting and no setup – not even seeing what the values of `Month` are. This makes re-using this code with more data much easier

Looping over variables

The `by()` function provides loops over subsets of the data – in a functional way. But perhaps you want to loop over **variables** in a data frame, applying a function to each;

```
> apply(X=airquality, MARGIN=2, FUN=mean, na.rm=TRUE)
     Ozone    Solar.R       Wind       Temp      Month        Day
 42.129310 185.931507   9.957516  77.882353   6.993464  15.803922
```

- `X`: an array (usually a matrix, or data frame)
- `MARGIN`: the dimension over which we apply the function. For 2D objects, 1 loops over rows, 2 loops over columns
- `FUN`: the function to be applied
- Any other arguments are passed to `FUN` – so to `mean()`, here

As ever, choosing a sensible number of decimal places is up to you – as is choosing a sensible variable of which to the meant.

And yes, we could also do this with a for loop...

Looping over variables

There is a widespread belief that `apply()` is faster than a `for()` loop over the columns. This is false – but useful, since it encourages people to use `apply()`. Use your discretion.

New functions

Suppose you want the mean and standard deviation for each variable. One solution is to apply a new function;

```
apply(airquality, 2,
      function(x){ c(mean=mean(x, na.rm=TRUE), sd=sd(x, na.rm=TRUE)) }
)
        Ozone   Solar.R     Wind     Temp    Month      Day
mean 42.12931 185.93151 9.957516 77.88235 6.993464 15.80392
sd   32.98788  90.05842 3.523001  9.46527 1.416522  8.86452
```

The code

```
function(x){ c(mean=mean(x,na.rm=TRUE), sd=sd(x,na.rm=TRUE)) }
```

translates into English as:

If you give me a vector, which I will call `x`*, I will* `mean` *it and* `sd` *it and give you the results in a named vector.*

As with `replicate()`, the last object evaluated is what the function returns, and for one-command functions you don't need curly brackets – but should use them anyway.

New functions

As we saw before, functions are objects, so they can be given names – which makes re-use easier, as well as making the code more readable;

```
mean.and.sd <- function(x){ c(mean=mean(x,na.rm=TRUE),
                              stddev=sd(x,na.rm=TRUE))
                          }
apply(airquality, 2, mean.and.sd)
```

Just like R's functions, yours can take more than one argument – just use e.g. `function(x,y,z,potato){}`

To pass arguments through to other functions, there's a special R-specific syntax, using ellipsis, i.e. "..." to represent "and anything else";

```
mean.and.sd2 <- function(x, ...){ c(mean=mean(x, ...), sd=sd(x, ...) ) }
apply(airquality, 2, mean.and.sd2, na.rm=TRUE)
```

Debugging code written using ellipsis can be tricky – did the error occur in `mean()` or `sd()`? – so use this trick cautiously.

by() revisited

With our own functions, we can use by() more generally;

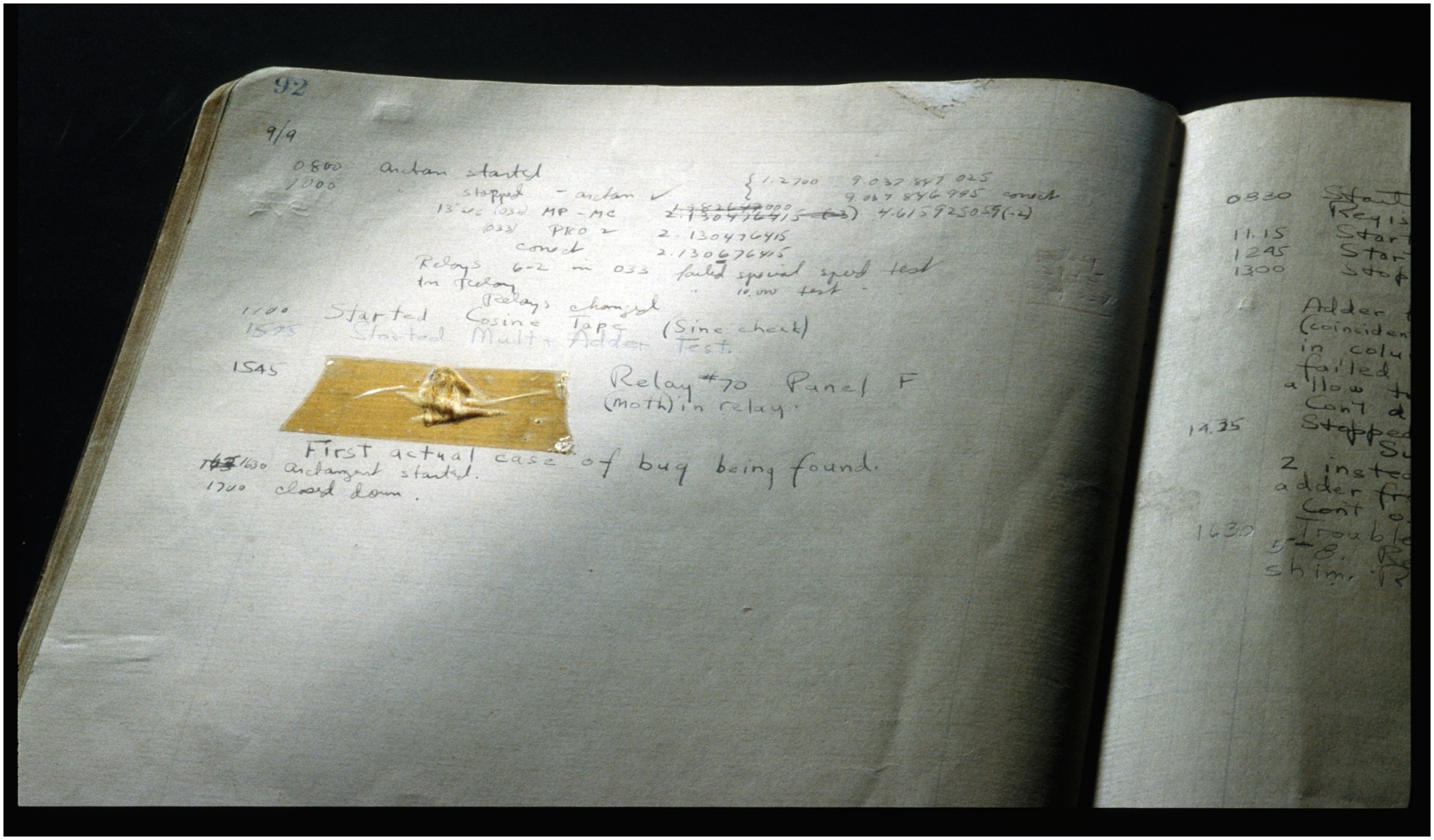
```
by(airquality, list(toohot=airquality$Temp>85),
      function(subset){round( apply(subset, 2, mean.and.sd), digits=2)} )
toohot: FALSE
     Ozone Solar.R  Wind  Temp Month   Day
mean 30.88  176.53 10.59 74.50  6.83 16.30
sd   27.25   95.42  3.41  7.78  1.49  8.58
------------------------------------------------------------
toohot: TRUE
     Ozone Solar.R Wind  Temp Month   Day
mean 79.22  219.44 7.73 89.74  7.56 14.06
sd   20.90   57.15 3.01  3.18  0.93  9.74
```

```
function(subset){ round(apply(subset, 2, mean.and.sd), digits=2) }
```

translates as

> *If you give me a data frame, which I will call* `subset`*,*
> *I will apply the* `mean.and.sd` *function to each variable,*
> *round to 2 decimal places, and give you the results.*

Debugging (again)



World's first computer code bug (1947) – discovered by debugger Grace Hopper – the bug is now stored in the Smithsonian.

Debugging (again)

Last week we used `traceback()` to suggest where problems might be occuring. Now we have to debug our own functions.

A simple example first, to show you the tools. Suppose* you wrote a function to calculate factorials, i.e. for integer x

$$x! = \prod_{i=1}^{x} i = 1 \times 2 \times 3 \ldots \times x$$

This can be done in R with a *recursive* function;

```
myfact <- function(x=8){
   if(x==1) return(x) else return(x*myfact(x-1))
}
> myfact() # using the default value
[1] 40320
> myfact(6)
[1] 720
```

* ...unaware of the `factorial()` function

Debugging (again)

What happens if our code had a bug?

```
mybadfact <- function(x){
   if(x==1) return(sqrt("hello")) else return(x*mybadfact(x-1))
}

> mybadfact(6)
Error in sqrt("hello") : non-numeric argument to mathematical function
> traceback()
6: mybadfact(x - 1) at #2
5: mybadfact(x - 1) at #2
4: mybadfact(x - 1) at #2
3: mybadfact(x - 1) at #2
2: mybadfact(x - 1) at #2
1: mybadfact(6)
```

Thinking about calls several steps "deep" in the call stack can be a challenge. It would be easier to go see what R was doing when those calls occurred.

R has an interactive debugger, to do just this – once turned on, when an error occurs it lets you interact with the call stack.

Debugging (again)

Finally;

```
> options(error=recover) # turn on the debugger
> mybadfact(6)           # the buggy command, leading to an error
Error in sqrt("hello") : non-numeric argument to mathematical function

Enter a frame number, or 0 to exit

1: mybadfact(6)
2: #2: mybadfact(x - 1)  # reverse order from traceback()
3: #2: mybadfact(x - 1)
4: #2: mybadfact(x - 1)
5: #2: mybadfact(x - 1)
6: #2: mybadfact(x - 1)

Selection: 2             # Enter a frame
Called from: mybadfact(x - 1)
Browse[1]> ls()          # What objects does R know about during this call?
[1] "x"
Browse[1]> x             # What is x's value here?
[1] 5
Browse[1]> c             # close this frame, back to the menu
```

Debugging (again)

... continued;

```
Enter a frame number, or 0 to exit

1: mybadfact(6)
2: #2: mybadfact(x - 1)
3: #2: mybadfact(x - 1)
4: #2: mybadfact(x - 1)
5: #2: mybadfact(x - 1)
6: #2: mybadfact(x - 1)

Selection: 6        # the troublesome frame
Called from: eval(substitute(browser(skipCalls = skip), list(skip = 7 - which)),
    envir = sys.frame(which))
Browse[2]> ls()
[1] "x"
Browse[2]> x        # now what's the value of x?
[1] 1
Browse[2]> c
```

... so looking at `mybadfact()` with input `x=1` should suggest what the problem is.

Debugging (again)

Before getting to work on `mybadfact()`...

```
Enter a frame number, or 0 to exit

1: mybadfact(6)
2: #2: mybadfact(x - 1)
3: #2: mybadfact(x - 1)
4: #2: mybadfact(x - 1)
5: #2: mybadfact(x - 1)
6: #2: mybadfact(x - 1)

Selection: 0
> options(error=NULL)  # Turn it off! Turn it off! Turn it off!!!
```

Going into interactive debugging mode every time you omit a parenthesis is incredibly irritating.

Debugging (again)

Now for a less-trite example; suppose you are simulating to investigate how much sample standard deviations can vary, for random Normal Y and random binary X. In particular, are two sample standard deviations more than 0.5 units apart in this example;

```
do.one <- function(n){                    # sample size per group
   x <- rbinom(2*n, size=1, prob=0.5) # group labels: 0/1, each with 50% probabili
   y <- rnorm(2*n, mean=0, sd=1)      # Random normal outcomes
   a <- by(y,x,sd)                    # see earlier slides!
   if( abs(diff(a))>0.5){ bigdiff <- TRUE } else {bigdiff <- FALSE}
   bigdiff
}

> set.seed(4)
> do.one(10)
[1] FALSE
> table( replicate(1000, do.one(10) ) )
FALSE  TRUE
  873   127
> table( replicate(1000, do.one( 5) ) )
Error in if (abs(diff(a)) > 0.5) { :
  missing value where TRUE/FALSE needed   # What's wrong?
```

Debugging (again)

Now using the debugger – investigate the last frame for which you understand the code;

```
Enter a frame number, or 0 to exit

1: table(replicate(1000, do.one(5)))
2: replicate(1000, do.one(5))
3: sapply(integer(n), eval.parent(substitute(function(...) expr)), simplify =
4: lapply(X = X, FUN = FUN, ...)
5: FUN(c(0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
6: do.one(5)
Selection: 6
```

As we wrote `do.one()`, we **should** understand what it's doing...

Debugging (again)

The debugger tells use what R is doing inside `do.one()`, when the error occured;

```
Called from: eval(substitute(browser(skipCalls = skip), list(skip = 7 - which)),
    envir = sys.frame(which))
Browse[1]> ls()
[1] "a" "n" "x" "y"
Browse[1]> x
 [1] 0 0 0 0 0 0 1 0 0 0
Browse[1]> y
 [1]  0.28855162  0.74701849  0.23138416 -1.35581032  0.96765873  0.14975207
 [7]  0.97566296  0.77858182  1.18455009 -0.08818449
Browse[1]> a
x: 0
[1] 0.7559985
------------------------------------------------------------
x: 1
[1] NA
Browse[1]> abs(diff(a))
[1] NA
```

The code halts because feeding an `NA` to `if()`, R doesn't know whether the condition holds, so it doesn't know which expression to run.

Debugging (again)

Other debugging tools;

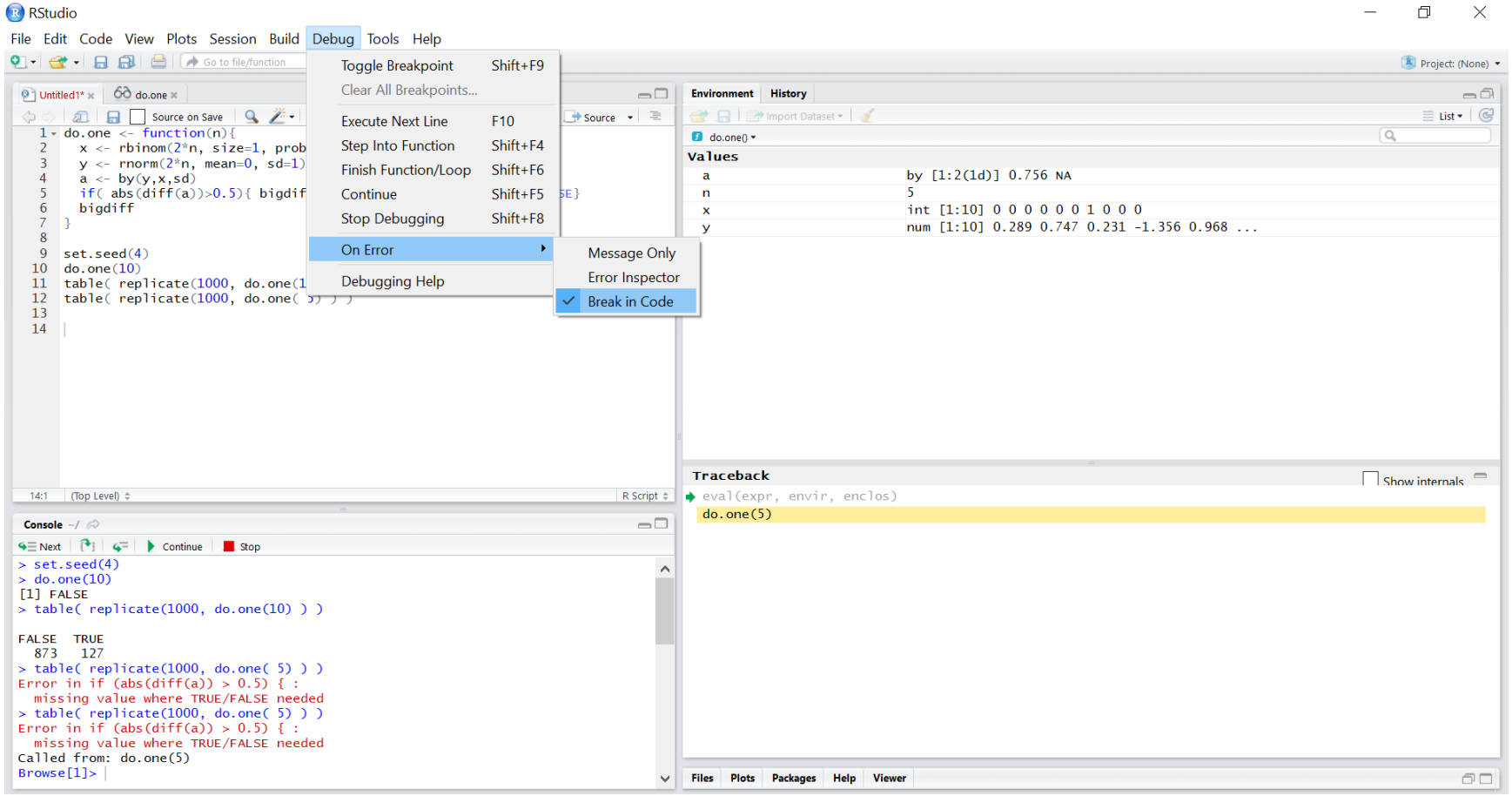
- Use `debug()` to start the browser when a function is called, or use `browser()` within a function to start the browser at that point
- Check all aspects of your code on simple examples **where you know the answer**, at least to reasonable accuracy
- Think **before** you program!

Some general *defensive programming* tips for avoiding errors;

- Check function inputs and give a `warning()` if the code will do something unexpected
- Check inputs to `if()` and the ranges in `for()` loops;
- Provide reasonable default arguments ...like my examples didn't
- `stop()` execution based on checks and give an informative error message

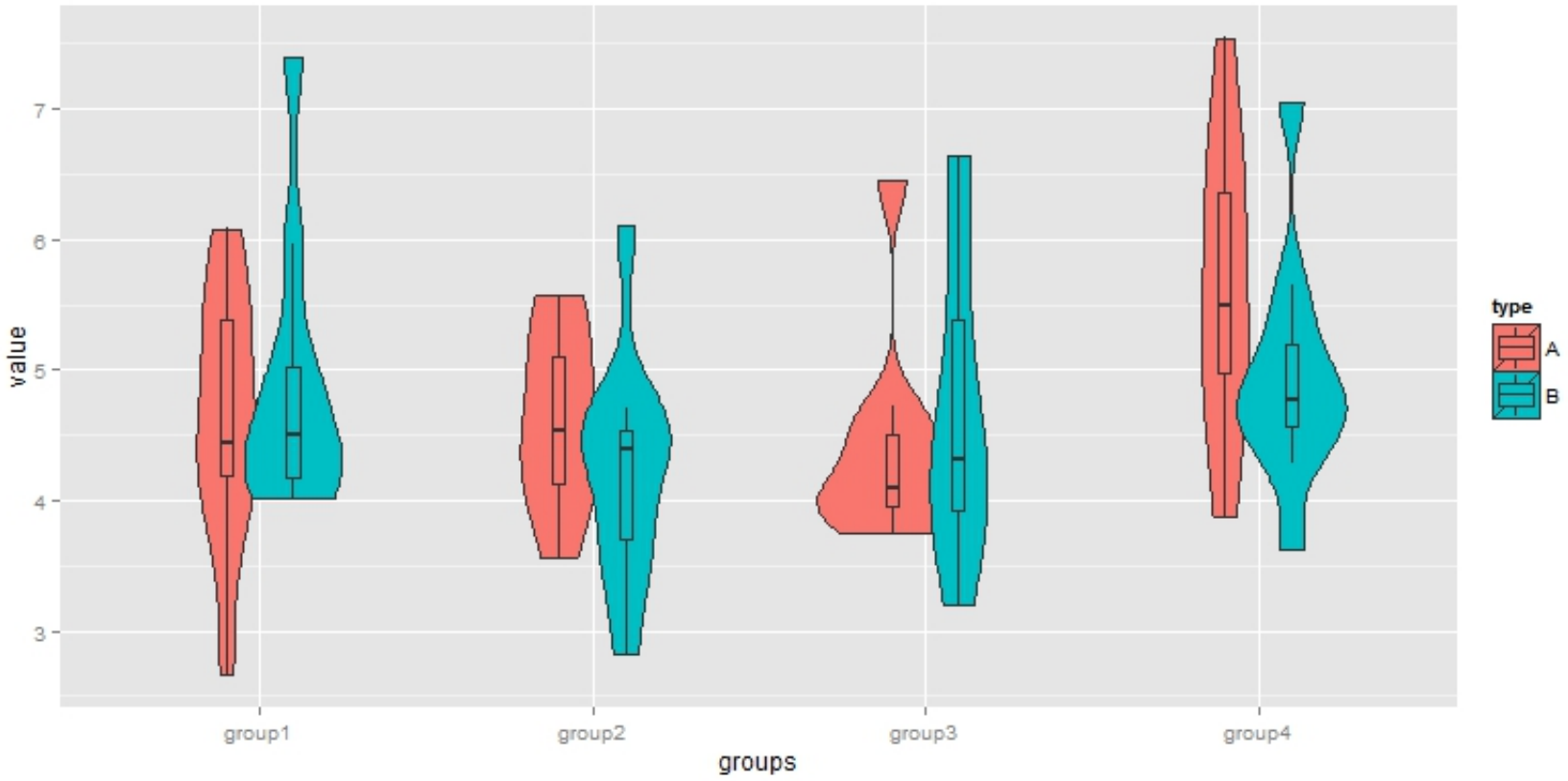
Debugging (again)

RStudio has nicer interfaces to the debugging tools;



Next time

- ggplot – with Ali Shojaie



Check your F-wing mailbox for HW1 return.

Appendix

Yes, R can do `while` and `repeat` loops;

```
i <- 1
my.mat <- matrix(NA, n, 3)
while(i <= n){
  z <- work.on.object(i)
  my.mat[i,] <- summary(z)
  i <- i+1
  }

i <- 1
my.mat <- matrix(NA, n, 3)
repeat{
  z <- work.on.object(i)
  my.mat[i,] <- summary(z)
  i <- i+1
  if(i>=n) break
  }
```

- Like `for()` loops, these do have a place – you may need them for some jobs
- Don't grow the output here either
- See `?Control` for the help page