# Assignment 2 Feedback

STA 242

May 9, 2015

## *Formatting Requirements*

Please make sure you do the following for every assignment:

- Include the URL of your Bitbucket repository.
- Put margins on every page, including source code.[1]
- Use a white or light-colored background for source code.
- Format tables neatly (no raw output).
- Put your files in the `AssignmentN` directory.[2]

[1] Otherwise there's nowhere to write feedback.

[2] For instance, Assignment 4 goes in the `Assignment4/` directory.

## *Using git*

Commit source code to your repository rather than archives such as `.tar.gz` or `.zip`. Git is designed for tracking source code, and using archives makes it harder for git to do its job properly. Archives also slow down anyone cloning your repository—they have to unpack the archive before they can look at your code.

Avoid committing data created by your code. If someone wants that data, they can run your code. Even pseudorandom data can be reproduced, as long as you make sure to set a seed in your code.

## *Code Design*

### *Lazy Default Arguments*

A lot of people tried to provide two different ways of calling `createBMLGrid()`, depending on whether a `density` or `ncars` parameter was set. An elegant way to do this is to use lazy default arguments:

```
createBMLGrid = function(rows, cols, ncars = rows * cols * density, density) {
    # ...
}
```

In this example, if the user doesn't specify `ncars`, it will be calculated from the other arguments. The calculation is *lazy*, because it only happens if you actually use `ncars` in your code.

If you need more than one line of code to set up a missing argument, check out the `missing()` function.

### *Keeping `if` Simple*

This code is from a student's `createBMLGrid()` function:

```
if( r<= 0 || c <=0 || ncars <=0){
  stop(" All the arguement must be positive!")
}
else{
  if( sum(ncars) >= r*c){
    stop(" Number of cars must be smaller than the number of grid cells!")
  }
  else{
    # ...
```

The code does an excellent job of checking for invalid input. However, the nested `if` statements make it a little hard to read.

When the `if` condition is true, R will see `stop()` and return from the function immediately. That means that code after the `if` statement will never run unless the `if` condition is false, so we can avoid using `else`. By using this idea, we can make the code easier to read:

```
if(r<= 0 || c <=0 || ncars <= 0)
  stop(" All the arguement must be positive!")

if(sum(ncars) >= r*c)
  stop(" Number of cars must be smaller than the number of grid cells!")

# ...
```

*Local Variables (Caching)*

The code below moves cars to the right, wrapping them around if they're in the last column of the grid.

```
rule = findMov - (ncol(grid)-1)*nrow(grid)
findMov[rule>0] =  findMov[rule>0] - (ncol(grid)-1)*nrow(grid)
findMov[rule <= 0] = findMov[rule <= 0] + nrow(grid)
```

In this code, `nrow(grid)` appears three times, and `ncol(grid)` appears twice. The dimensions of `grid` don't change, so it's more efficient to save them in a local variable. We can also improve the code by changing `rule` to the actual condition for being in the last column, and then giving it a more descriptive name. After making these changes and adding a comment to explain what's happening, we have:

```
rows = nrow(grid)
lastCol = (ncol(grid) - 1) * rows
inLastCol = (findMov - lastCol) > 0

# Wrap around cars in last column.
findMov[inLastCol] =  findMov[inLastCol] - lastCol

findMov[!inLastCol] = findMov[!inLastCol] + rows
```

*Combining Functions (Parameterization)*

The student who wrote the two functions in this example put most of the differences in the subfunctions `BlueTmp()` and `RedTmp()`. This makes it especially easy to combine them into a single function.

```
BlueReal=
  function(r,blue,red)
  {
    tmp=BlueTmp(blue,r)
    jammed=tmp %in% c(red,blue)
    tmp[jammed]=blue[jammed]
    velocity=1-sum(jammed)/length(blue)
    list(blue=tmp, velocity=velocity)
  }

RedReal=
  function(r,total,blue,red)
  {
    tmp=RedTmp(red,r,total)
    jammed=tmp %in% c(red,blue)
    tmp[jammed]=red[jammed]
    velocity=1-sum(jammed)/length(red)
    list(red=tmp, velocity=velocity)
  }
```

You might still be uncertain of what these functions actually do—lack of comments and nondescript variable names like `tmp` makes it hard to tell. We can fix that as well.

```
move = function(rows, total, movers, nonmovers, move_cars)
    # Move cars, checking for jams.
    #
    # Args:
    #   rows        number of rows
    #   total       total number of cars
    #   movers      moving cars
    #   nonmovers   stationary cars
    #   move_cars   function to move the cars
{
    moved = move_cars(movers, rows, total, move_cars)

    # Move back jammed cars.
    jammed = moved %in% c(movers, nonmovers)
    moved[jammed] = movers[jammed]

    velocity = 1 - sum(jammed) / length(movers)

    list(moved=moved, velocity=velocity)
}
```

The new version uses a parameter, `move_cars`, to control whether `RedTmp()` or `BlueTmp()` is used (this requires that they have identical

parameters).