Demography 213
Loops and branches with dataframes and **avoiding** loops and branches with tapply()

Carl Mason
carlm@demog.berkeley.edu

September 19, 2016

Abstract

This week we continue with our exploration of R's looping and branching capabilities. We will (re)introduce two fancy new data structures (***lists*** and ***dataframes***) and we will catch a glimpse of the ***tapply()*** and ***sapply()*** functions – fiendishly clever functions that often lets you *avoid* using loops and branches.

The `demonstration.r` program, which we will mostly walk through in class, is a sneaky way of introducing you to Socsim, a microsimulation program (that is full of dissertations) while talking about data frames and functions.

Contents

1 Introduction **1**

2 Exercise **2**

2.1 "Small" Problems with even smaller answers 3

1 Introduction

It will come as no surprise that there are many kinds of population simulations. Our Leslie Matrix simulation would be called *a deterministic macro-simulation* because there was nothing random about it and it dealt with counts of organisms rather than individuals. Had we allowed the fertility and/or mortality rates to change randomly from period to period the simulation would have become a "stochastic *macro*-simulation" because it had randomness (stochasticity) but also dealt with cohorts of organisms rather than individuals.

But of course population simulation can be even more complicated. ***Stochastic micro-simulations*** track (simulated) individuals with (simulated) life course events governed by stochastic processes. Our

favorite demographic micro-simulation program is SOCSIM, a program developed in this department a very long time ago. Since then it has been used by world famous demographers, talented graduate students, and talented graduate students who have gone on to become world famous demographers.

Using SOCSIM directly would be prohibitively boring right now, so we will "simply"[1] spend our valuable time examining the output from some else's SOCSIM experiment.

SOCSIM results consist of life histories of huge numbers of fictitious "people" who "lived" generally over a very long period of fictitious time. SOCSIM experiments generally start with a small initial set of unrelated people and then as simulated time passes, these people live, marry, give birth, migrate (sometimes) and die. Over the course of many generations, the result is a large list of "people" with linkages to **all** of their ancestors. In other words, SOCSIM produces a complete kinship network for an entire population – with **no** reporting errors and **no** missing data. Clearly, this is fantastic resource for Demographers – with the one drawback that it is not a population that ever existed. But if such non-reality "issues" seem daunting to you now, it is only because you have not yet spent sufficient time in academia.

The `demonstration.r` program includes an investigation of the *marriage squeeze* that might have happened in a country that might have been Slovakia during late twentieth century. Marriage squeezes occur when the balance between the sexes is uneven. Wars, sex selective abortion, migration, and heavy drinking can cause marriage squeezes, but in the case of Slovakia, the cause "was"[2] declining fertility combined with preferences for older husbands and younger wives.

2 Exercise

This week's exercise consists of the usual `demonstration.r` program – which we will introduce in class and several small questions presented below with answers in bacterial sized font. This time it will be more useful to do the small problems first and then attack the `demonstration.r` file.

So the drill for this week is:

1. **Read/Review Chapters 3-6 in *Introduction to R*** (Pages 19-32) You can skim lightly over sections 5.7.3 - 5.7.5 which deal with eigen values, Single value decomposition and matrix inversion. Knowing that these features exist could win you money on a game show someday – but we won't be using them in 213.

2. **become aware of *Socsim Oversimplified***

   SOCSIM's homepage is `http://lab.demog.berkeley.edu/socsim`. You'll want to explore this site, in more detail, later in your career as you become desperate for a dissertation topic. For now just take a look at the chapter on ***population files*** in the document called *Socsim Oversimplified*. You will be working with files of this kind this week.

[1] the word "simply" is used here ironically

[2] many historical facts were ignored in the interest of a better story

3. **Setup your Rstudio project for the current week** EVEN though you are going to do the short problems below first ... it most convenient to copy the demonstration.r file before you launch Rstudio:

   - `@:> cp ~carlm/213/DataStructure/demonstration.r 213/WeekN`
   - launch Rstudio in the browser of your choice.
   - Create a New Project in and Existing Directory...

4. **attack the "small" problems below** with the usual thirst for knowledge for which young demographers are well known. In a new cleverly named and .R suffixed file in your current week's Rstudio project...try to do the exercises before looking at the solutions (which are of course provided in unreadable type – if you can't then at least try to understand why the solutions makes sense – or better yet – find your instructor's mistakes and gloat.

5. **Apply the skills you have just acquired to the questions in the demonstration.r** file that you previously coppied into your current week's Rstudio project.

6. **email cmason@berkeley.edu** with news of your success and excitement over the new skills that you have acquired.

2.1 "Small" Problems with even smaller answers

The small problems below use a simple data.frame with data on eruptions and waiting times between eruptions of a geyser in Yellow Stone National Park. The exercises below should help you get familiar with slinging around small dataframes before you tackle the larger ones in `demonstration.r` – which you will do later.

Solve the problem below in a creatively named file in your weekN project folder.

1. get the "faithful" data set into your workspace by typing `data(faithful)` then figure out what "class" of object `faithful` is.

```
data(faithful)
attributes(faithful)


## $names
## [1] "eruptions" "waiting"
##
## $row.names
##   [1]   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17
##  [18]  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34
##  [35]  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51
##  [52]  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68
##  [69]  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85
##  [86]  86  87  88  89  90  91  92  93  94  95  96  97  98  99 100 101 102
## [103] 103 104 105 106 107 108 109 110 111 112 113 114 115 116 117 118 119
## [120] 120 121 122 123 124 125 126 127 128 129 130 131 132 133 134 135 136
## [137] 137 138 139 140 141 142 143 144 145 146 147 148 149 150 151 152 153
## [154] 154 155 156 157 158 159 160 161 162 163 164 165 166 167 168 169 170
## [171] 171 172 173 174 175 176 177 178 179 180 181 182 183 184 185 186 187
## [188] 188 189 190 191 192 193 194 195 196 197 198 199 200 201 202 203 204
```

```
## [205] 205 206 207 208 209 210 211 212 213 214 215 216 217 218 219 220 221
## [222] 222 223 224 225 226 227 228 229 230 231 232 233 234 235 236 237 238
## [239] 239 240 241 242 243 244 245 246 247 248 249 250 251 252 253 254 255
## [256] 256 257 258 259 260 261 262 263 264 265 266 267 268 269 270 271 272
##
## $class
## [1] "data.frame"
```

2. Execute and interpret the following R code:

```
sum(faithful[,"eruptions"] != faithful$eruptions)
```

```
## [1] 0
```

```
sum(faithful[,1] != faithful$eruptions)
```

```
## [1] 0
```

```
sum(faithful[["eruptions"]] != faithful$eruptions)
```

```
## [1] 0
```

```
sum(faithful[[1]]!=faithful$eruptions)
```

```
## [1] 0
```

data.frames (like `faithful`) are lists and as such you can select from them using the list syntax $ as well as the "[[ ]]" double bracket syntax. Data.frames can also be treated as arrays (which as you know are special kind of `vector`, there are, therefore generally 3 different ways to select a column from a data.frame).

3. Find the mean waiting time between the first 20 eruptions (rows) in `faithful`

```
mean(faithful$waiting[1:20])
```

```
## [1] 69.65
```

```
## or
mean(faithful[1:20,"waiting"])
```

```
## [1] 69.65
```

```
## or
mean(faithful[1:20,2])
```

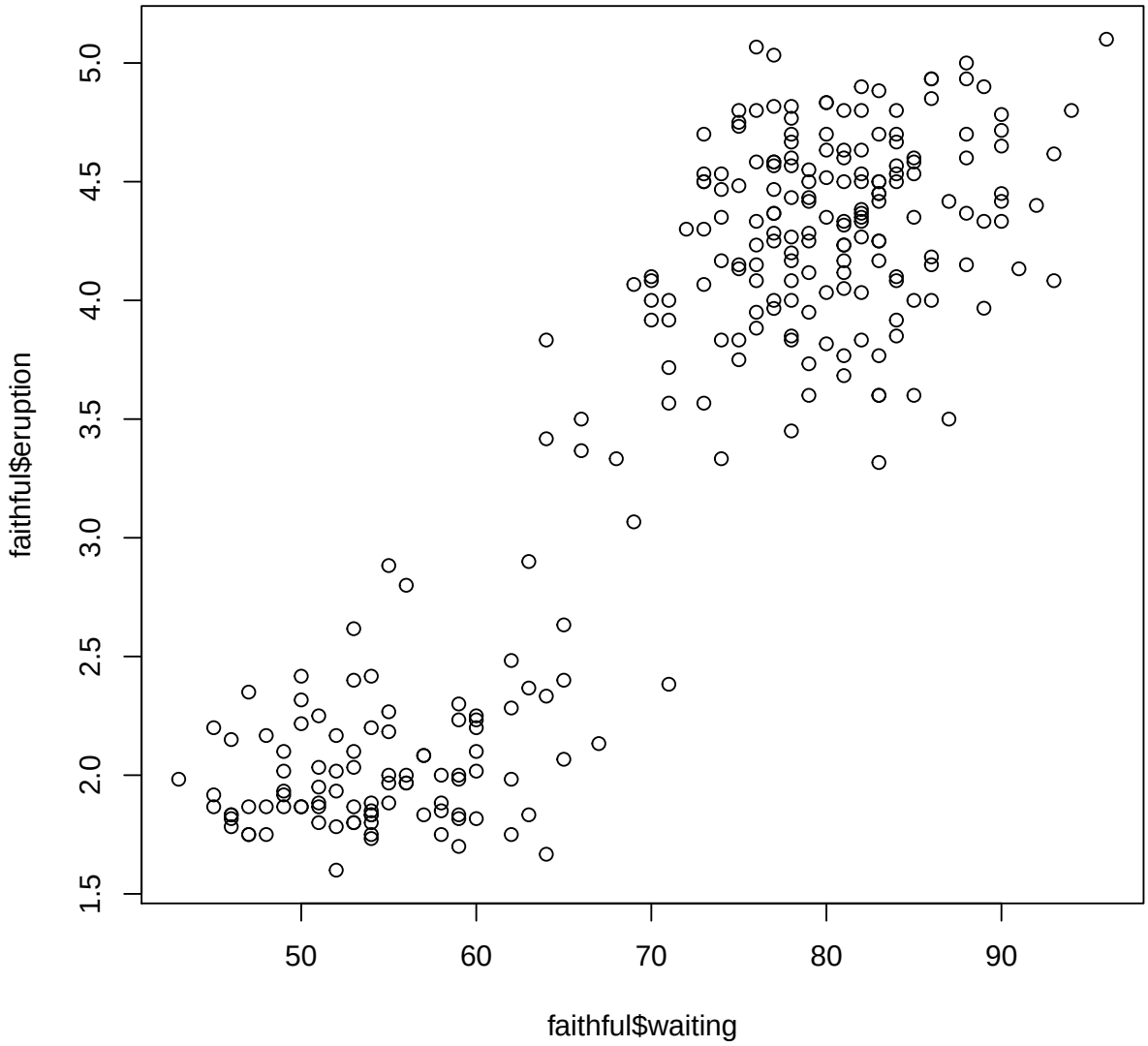
```
## [1] 69.65
```

```
## or
mean(faithful[["waiting"]][1:20])
```

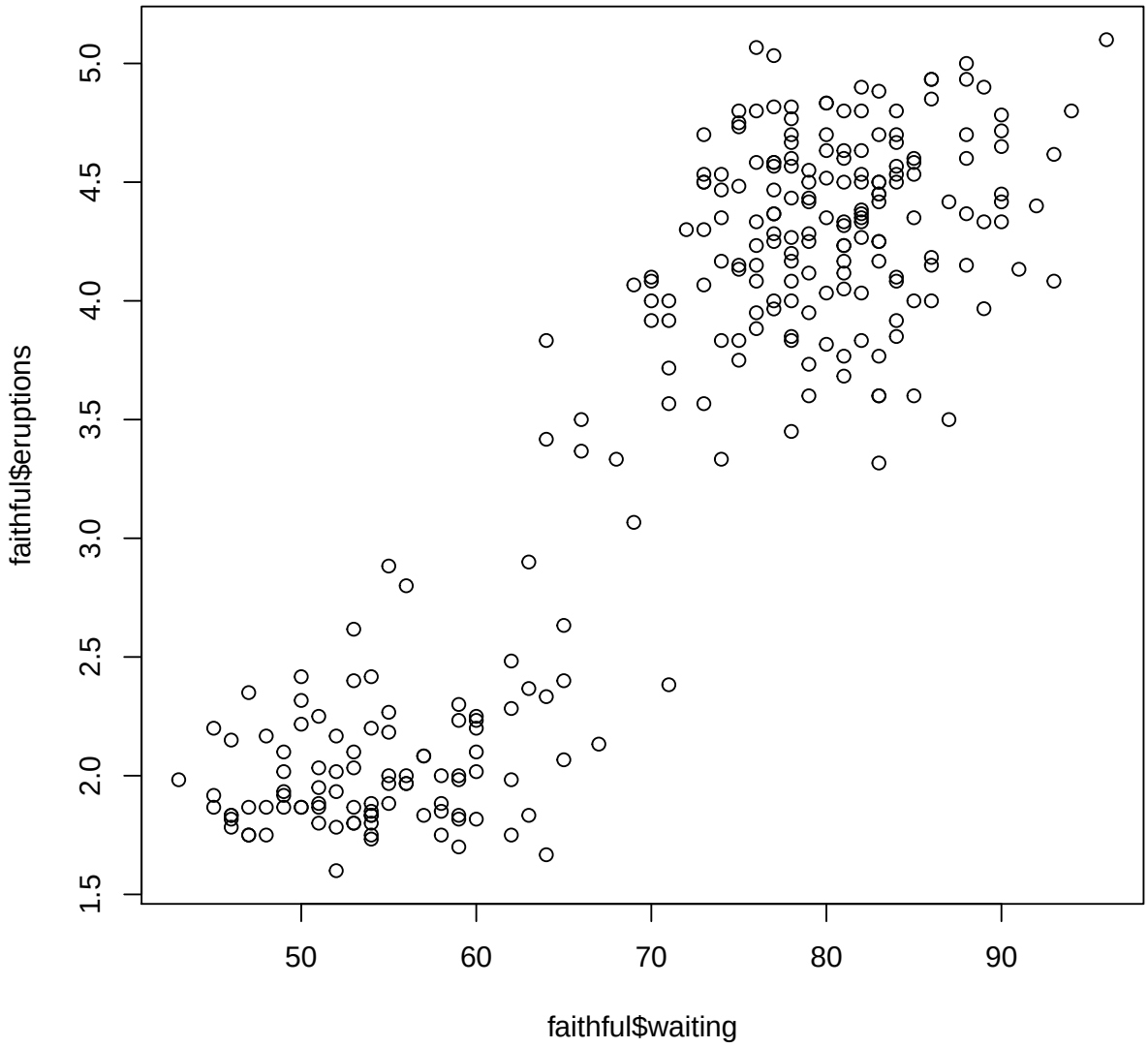
```
## [1] 69.65
```

4. Draw a scatter plot of eruption duration vs waiting time.

```
plot(faithful$eruption ~ faithful$waiting)
```

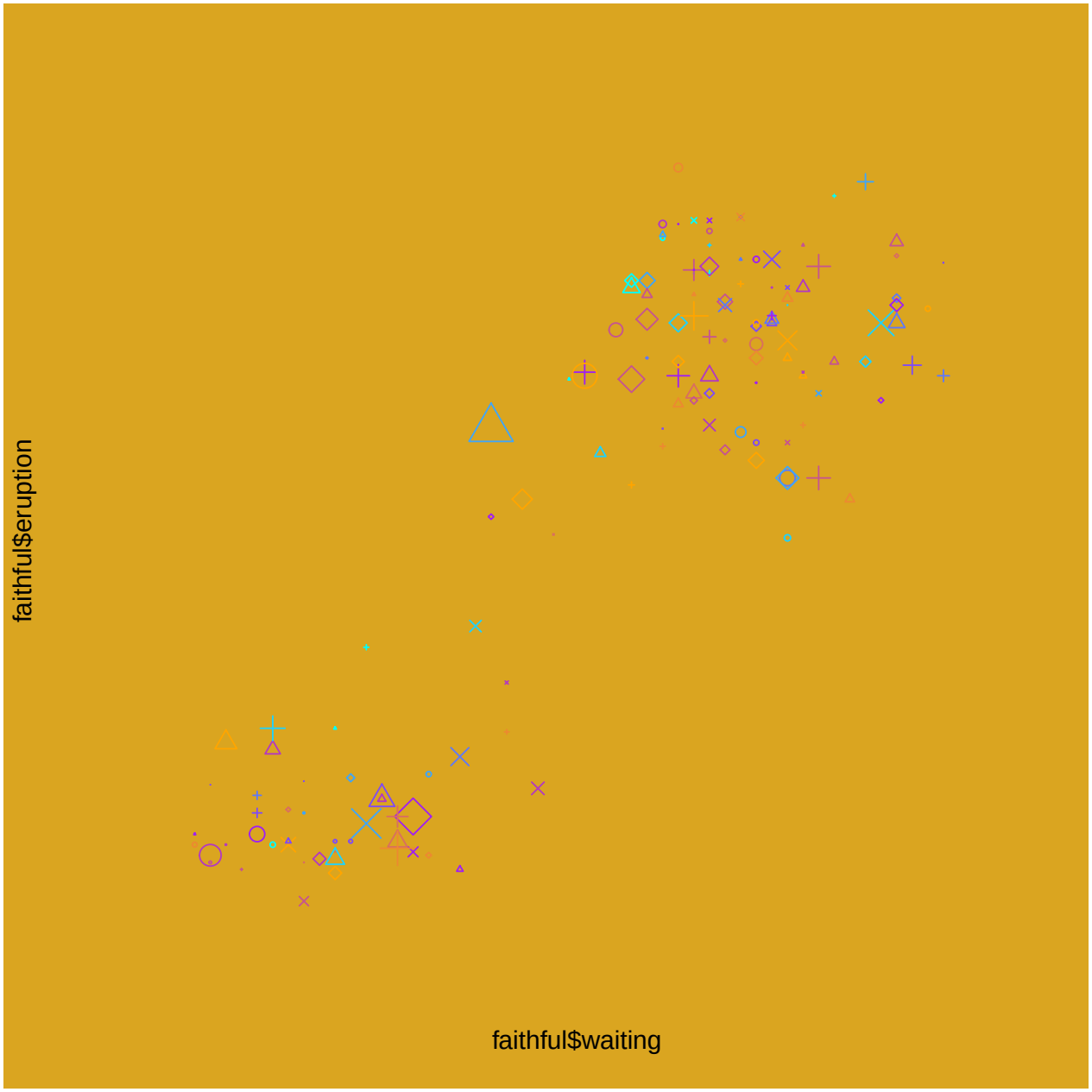



```
## or
plot(y=faithful$eruptions ,x=faithful$waiting)
```



OR...

```
par(bg='goldenrod')
plot(faithful$eruption ~ faithful$waiting,
     cex=rnorm(nrow(faithful)),
     col=colorRampPalette(c("cyan","purple","orange"))(11),
     pch=1:5,axes=FALSE)
```



5. Add a third column to the `faithful` dataframe. Call it logwait and let it contain the natural logs of faithful$waiting.

```
faithful$logwait<-log(faithful$waiting)
## verify that it is what we think it is
sum(log(faithful$waiting) != faithful$logwait)
```

```
## [1] 0
```

6. Add another column to `faithful` that holds a ***logical*** value which indicates whether or not the corresponding eruption time is longer or shorter than the median.

```
faithful$longErups<- faithful$eruptions > median(faithful$eruption)
```

7. Explain why longErups (or whatever you called the column that you created in the previous question) looks different in the following two lines of code:

```
head(faithful)

##   eruptions waiting  logwait longErups
## 1     3.600      79 4.369448     FALSE
## 2     1.800      54 3.988984     FALSE
## 3     3.333      74 4.304065     FALSE
## 4     2.283      62 4.127134     FALSE
## 5     4.533      85 4.442651      TRUE
## 6     2.883      55 4.007333     FALSE

head(as.matrix(faithful))

##   eruptions waiting  logwait longErups
## 1     3.600      79 4.369448         0
## 2     1.800      54 3.988984         0
## 3     3.333      74 4.304065         0
## 4     2.283      62 4.127134         0
## 5     4.533      85 4.442651         1
## 6     2.883      55 4.007333         0
```

8. create a new dataframe, called `faithfulS` which consists of all of the elements in `faithful` but with the rows ordered from longest to shortest waiting time. HINT check out the function `order()`

```
faithfulS<- faithful[order(faithful$waiting,decreasing=TRUE),]
head(faithfulS)

##     eruptions waiting  logwait longErups
## 149     5.100      96 4.564348      TRUE
## 218     4.800      94 4.543295      TRUE
## 158     4.083      93 4.532599      TRUE
## 170     4.617      93 4.532599      TRUE
## 66      4.400      92 4.521789      TRUE
## 203     4.133      91 4.510860      TRUE
```

9. Write a loop that will take the average waiting time associated long eruptions and short eruptions where long and short eruptions are defined as above.

```
res9<-rep(0,length=2)  ## a vector to hold the two results
for(i in 1:nrow(faithful)){
  if(faithful$longErups[i] == TRUE){
    res9[1]<-res9[1]+ faithful$waiting[i]
  }else{
    res9[2]<-res9[2]+ faithful$waiting[i]
  }
}
res9/c(sum(faithful$longErups),
      sum(!faithful$longErups))


## [1] 81.02273 61.35000
```

10. Find the same two average values as above using an `ifelse()`

```
res10<-rep(0,length=2)
res10[1]<-mean(ifelse(faithful$longErups==TRUE,faithful$waiting,NA),na.rm=T)
res10[2]<-mean(ifelse(faithful$longErups==FALSE,faithful$waiting,NA),na.rm=T)
res10


## [1] 81.02273 61.35000
```

11. Find the same two values as above but using instead the `tapply()` function.

```
tapply(faithful$waiting, faithful$longErups,mean)


##    FALSE     TRUE
## 61.35000 81.02273
```

12. list objects. Lists in R are called ***recursive*** objects because they can contain other objects including other lists. Big deal huh?. What lists are good for is keeping a bunch of different things together. For example when you run statistical estimation function such as `lm()` (regression) the result is returned as a list. Chiefly this is because the result of `lm()` consists of objects of different shapes and sizes:

```
lm.res<-lm(eruptions~waiting,data=faithful)
names(lm.res)


##  [1] "coefficients"  "residuals"     "effects"       "rank"
##  [5] "fitted.values" "assign"        "qr"            "df.residual"
##  [9] "xlevels"       "call"          "terms"         "model"


##   one cannot make an array
##   out of a vector of coefficient estimates and a vector of fitted
##   values, and a list of independent variables and so on.  If one were
##   to simply drop all these objects independently into the
##   workspace... only bad things could happen.
```

lists can be references with "$" notation (dataframes are also lists) or they can be referenced with .. "[[ ]]" **double square brackets**

```
  ## reference by name of the list element
  lm.res$coefficients
```

```
## (Intercept)     waiting
## -1.87401599  0.07562795
```

```
  ## reference by index and [[]]
  lm.res[[1]]
```

```
## (Intercept)     waiting
## -1.87401599  0.07562795
```

```
  ## reference by name in [[]]
  lm.res[["coefficients"]]
```

```
## (Intercept)     waiting
## -1.87401599  0.07562795
```

```
  ## an element of a list element:
  lm.res$coefficients[2]
```

```
##    waiting
## 0.07562795
```

```
  ## or
  lm.res[[1]]["waiting"]
```

```
##    waiting
## 0.07562795
```

```
  ## or
  lm.res[["coefficients"]][2]
```

```
##    waiting
## 0.07562795
```

How might we separate `faithful` into a 4 element list where each element holds a subset of rows of `faithful` depending on whether `waiting` or `eruptions` are greater or less than the their median?

```
  ## First the hard way:
faithful$whatsort<-
  paste(
        ifelse(faithful$eruptions > median(faithful$eruptions),"H","L"),
        ifelse(faithful$waiting > median(faithful$waiting),"H","L"),sep='')


faithful.list<-list()
faithful.list$HH<-faithful[faithful$whatsort == "HH",]
faithful.list$HL<-faithful[faithful$whatsort == "HL",]
faithful.list$LH<-faithful[faithful$whatsort == "LH",]
faithful.list$LL<-faithful[faithful$whatsort == "LL",]

table(faithful$whatsort)


##
##  HH  HL  LH  LL
## 105  27  24 116


dim(faithful.list$HH)


## [1] 105   5


dim(faithful.list[[1]])


## [1] 105   5


## Now the EASY way:
## same as above -- create an indicator variable
faithful$whatsort<-
    paste(
        ifelse(faithful$eruptions > median(faithful$eruptions),"H","L"),
        ifelse(faithful$waiting > median(faithful$waiting),"H","L"),sep='')


faithful.list2<-split(x=faithful,f=faithful$whatsort)
dim(faithful.list2$HH)


## [1] 105   5
```

That's it for Old Faithful, please turn now to the `demonstration.r`.