Lecture 3: More Data Frames, and Flow Control

Statistical Computing, 36-350

Monday September 14, 2015

Outline

- Making and working with data frames
- Conditionals: switching between different calculations
- Iteration: doing something over and over
- Vectorizing: avoiding explicit iteration

In our last thrilling episode

- Vectors: series of values all of the same type, e.g., `v[5]`, `v["name"]`
- Arrays: multi-dimensional generalization of vectors, e.g., `a[5,6,2]`, `a[,6,]`, `a["rowname", "colname"", "layername"]`
- Matrices: special 2d arrays with matrix math, e.g., `m[5,6]`, `m[,6]`, `m[,"colname"]`
- Lists: series of values of mixed types, e.g., `l[[3]]`, `l$name`
- Data frames: hybrid of matrix and list

Data frames, encore

- 2d tables of data
- Each case/observation is a row
- Each variable/feature is a column
- Variables can be of any type (numbers, text, Booleans, ...)
- Both rows and columns can be assigned names

Creating an example data frame

```
library(datasets)
states = data.frame(state.x77, Abbr=state.abb,
                    Region=state.region, Division=state.division)
```

Here `data.frame()` is combining a pre-existing matrix (`state.x77`), a vector of characters (`state.abb`), and two vectors of qualitative categorical variables (called **factors**; `state.region`, `state.division`)

Column names are preserved, or guessed if not explicitly set

```
colnames(states)
```

```
##  [1] "Population" "Income"     "Illiteracy" "Life.Exp"   "Murder"
##  [6] "HS.Grad"    "Frost"      "Area"       "Abbr"       "Region"
## [11] "Division"
```

```
states[1,]
```

```
##         Population Income Illiteracy Life.Exp Murder HS.Grad Frost  Area
## Alabama       3615   3624        2.1    69.05   15.1    41.3    20 50708
##         Abbr Region           Division
## Alabama   AL  South East South Central
```

Data frame access

By row and column index:

```
states[49,3]
```

```
## [1] 0.7
```

By row and column names:

```
states["Wisconsin","Illiteracy"]
```

```
## [1] 0.7
```

Data frame access (continued)

All of a row:

```
states["Wisconsin",]
```

```
##           Population Income Illiteracy Life.Exp Murder HS.Grad Frost  Area
## Wisconsin       4589   4468        0.7    72.48      3    54.5   149 54464
##           Abbr        Region           Division
## Wisconsin   WI North Central East North Central
```

(Exercise: what is the class of `states["Wisconsin",]`?)

Data frame access (continued)

All of a column:

```
head(states[,3])
```

```
## [1] 2.1 1.5 1.8 1.9 1.1 0.7
```

```
head(states["Illiteracy"])
```

```
## [1] 2.1 1.5 1.8 1.9 1.1 0.7
```

```
head(states$Illiteracy)
```

```
## [1] 2.1 1.5 1.8 1.9 1.1 0.7
```

Data frame access (continued)

Rows matching a condition:

```
states[states$Division=="New England", "Illiteracy"]
```

```
## [1] 1.1 0.7 1.1 0.7 1.3 0.6
```

```
states[states$Region=="South", "Illiteracy"]
```

```
##  [1] 2.1 1.9 0.9 1.3 2.0 1.6 2.8 0.9 2.4 1.8 1.1 2.3 1.7 2.2 1.4 1.4
```

Replacing values

Parts or all of the data frame can be assigned to:

```
summary(states$HS.Grad)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   37.80   48.05   53.25   53.11   59.15   67.30
```

```
states$HS.Grad = states$HS.Grad/100
summary(states$HS.Grad)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##  0.3780  0.4805  0.5325  0.5311  0.5915  0.6730
```

```
states$HS.Grad = 100*states$HS.Grad
```

with()

What percentage of literate adults graduated HS?

```
head(100*(states$HS.Grad/(100-states$Illiteracy)))
```

```
## [1] 42.18590 67.71574 59.16497 40.67278 63.29626 64.35045
```

`with()` takes a data frame and evaluates an expression "inside" it:

```
with(states, head(100*(HS.Grad/(100-Illiteracy))))
```

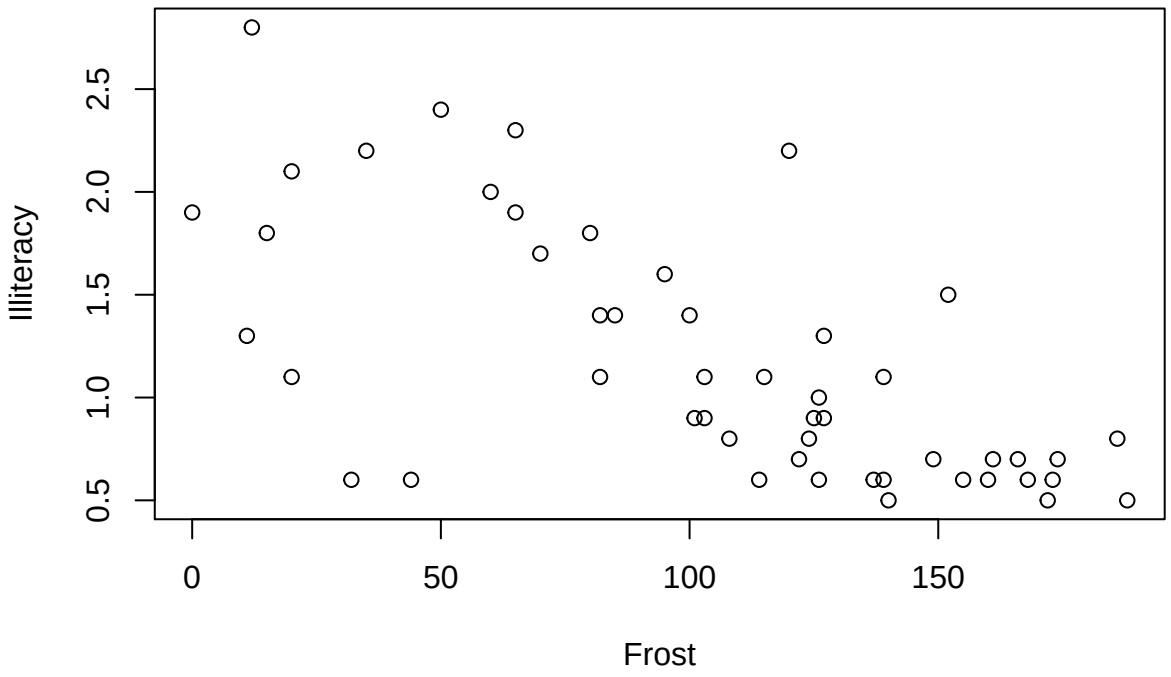
```
## [1] 42.18590 67.71574 59.16497 40.67278 63.29626 64.35045
```

Means that you can save writing `states$xxx` many times

Data arguments

Similar to the usage in `with()`, lots of functions take `data` arguments, and look variables up in that data frame:

```
plot(Illiteracy~Frost, data=states)
```



Conditionals and flow control

Have the computer decide what to do based on a condition. A mathematical example you all know:

$$|x| = \begin{cases} x & \text{if } x \geq 0 \\ -x & \text{if } x < 0 \end{cases}$$

Another more complex one:

$$\psi(x) = \begin{cases} x^2 & \text{if } |x| \leq 1 \\ 2|x| - 1 & \text{if } |x| > 1 \end{cases}$$

(Exercise: plot ψ in R)

`if()`

Simplest conditional:

```
if (x >= 0) {
  x
} else {
  -x
}
```

Condition in `if()` needs to give one `TRUE` or `FALSE` value

`else()` clause is optional

Single line actions don't need braces, as in:

```
if (x >= 0) x else -x
```

Nested `if()`

We can nest `if()` statements arbitrarily deeply:

```
if (x^2 < 1) {
  x^2
} else {
  if (x >= 0) {
    2*x-1
  } else {
    -2*x-1
  }
}
```

This can get ugly though

`switch()`

We can simplify a nested `if` with `switch()`: give a variable to select on, then a value for each option

```
switch(type.of.summary,
       mean=mean(states$Murder),
       median=median(states$Murder),
       histogram=hist(states$Murder),
       "I don't understand")
```

(Exercise: set `type.of.summary` to, succesively, "mean", "median", "histogram", and "mode", and explain what happens)

Combining Booleans: `&&` and `||`

`&` work `|` like `+` or `*`: the combine terms elementwise

But flow control always wants just one Boolean value. Hence we can skip calculating what's not needed

`&&` and `||` give just *one* Boolean, lazily:

```
(0 > 0) && (all.equal(42%%6, 169%%13))
```

```
## [1] FALSE
```

```
(0 > 0) && (MyCoolNewVariable == 0)
```

```
## [1] FALSE
```

(In both cases, R *never* evaluates the expression on the right)

In summary, use `&&` and `||` for conditionals, `&` and `|` for subsetting

Iteration

Repeat similar actions multiple times, as in:

```
table.of.logarithms = vector(length=7,mode="numeric")
table.of.logarithms
```

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

```
for (i in 1:length(table.of.logarithms)) {
  table.of.logarithms[i] = log(i)
}
table.of.logarithms
```

```
## [1] 0.0000000 0.6931472 1.0986123 1.3862944 1.6094379 1.7917595 1.9459101
```

`for()`

```
for (i in 1:length(table.of.logarithms)) {
  table.of.logarithms[i] = log(i)
}
```

`for` increments a **counter** (here `i`) along a vector (here `1:length(table.of.logarithms)`). It loops through the **body** (the expression inside the braces) until it runs through the vector

We call this "**iterating over** the vector"

The body of the `for()` loop

Can contain just about anything, including:

- `if()` statements
- other `for()` loops (nested iteration)

Nested iteration example

```
c = matrix(0, nrow=nrow(a), ncol=ncol(b))
if (ncol(a) == nrow(b)) {
  for (i in 1:nrow(c)) {
    for (j in 1:ncol(c)) {
      for (k in 1:ncol(a)) {
        c[i,j] = c[i,j] + a[i,k]*b[k,j]
      }
    }
  }
} else {
  stop("matrices a and b non-conformable")
}
```

`while()`: conditional iteration

Babylonian method for finding square root of a number x:

```
while (abs(x - r^2) > 1e-06) {
  r = (r + x/r)/2
}
```

Condition in the argument to `while()` must be a single Boolean value (like `if()`)

Body is looped over until the condition is `FALSE` (so can loop forever . . .)

Loop never begins unless the condition starts `TRUE`

`for()` versus `while()`

`for()` is better when the number of times to repeat (values to iterate over) is clear in advance

`while()` is better when you can recognize when to stop once you're there, even if you can't guess it to begin with

Every `for()` could be replaced with a `while()` (Exercise: show this)

`while(TRUE)` or `repeat`: unconditional iteration

`while(TRUE)` and `repeat`: both have the same function; just repeat the body indefinitely, unless something causes the flow to break

```
repeat {
  ans = readline("Who is the best Professor of Statistics at CMU? ")
  if (ans == "Tibs" || ans == "Tibshirani" || ans == "Ryan") {
    cat("Yes! You get an 'A'.")
    break
  }
  else {
    cat("Wrong answer!\n")
  }
}
```

Note that `break` gets us out of the loop; also, `next` skips the rest of the body, and starts us at the top of the loop again

Avoiding iteration

R has many ways of **avoiding iteration**, by acting on whole objects.
This is called **vectorization**

- It can be conceptually clearer
- It can lead to simpler code
- It is often faster (sometimes a little, sometimes drastically)

Vectorized arithmetic

How many languages add 2 vectors:

```
c = vector(length(a))
for (i in 1:length(a)) {  c[i] = a[i] + b[i]  }
```

How R adds 2 vectors:

```
a+b
```

Also recall our `for()` loop for matrix multiplication, versus the simple `a %*% b`

Vectorized functions

Many functions are set up to vectorize automatically

```
abs(-3:3)
```

```
## [1] 3 2 1 0 1 2 3
```

```
log(1:7)
```

```
## [1] 0.0000000 0.6931472 1.0986123 1.3862944 1.6094379 1.7917595 1.9459101
```

See also `apply()` from last week

We'll come back to this in great detail later

Vectorized conditions: `ifelse()`

```
ifelse(x^2 > 1, 2*abs(x)-1, x^2)
```

The first argument here is a Boolean vector. Then `ifelse()` picks from the second and third arguments as appropriate, when it encounters `TRUE` or `FALSE`

Summary

- Data frames: useful and ubiquitous
- Conditionals: `if()`, nested `if()`, `switch()`
- Iteration: `for()`, `while()`
- Avoiding iteration with whole-object ("vectorized") operations

What is truth?

- 0 counts as `FALSE`; other numeric values count as `TRUE`
- `T` and `F` count as `TRUE` and `FALSE` (unless you've defined variables with these names, to mean otherwise)
- The strings `"TRUE"` and `"FALSE"` count as you'd hope
- Most everything else gives an error

Advice: don't play games here; be sure control expressions are getting Boolean values

---

Conversely, in arithmetic, `FALSE` is 0 and `TRUE` is 1

```
mean(states$Murder > 7)
```

```
## [1] 0.48
```

Babylonian method of root finding

Look this up for a proper historical account!

Given: x, find $\sqrt{x}$. Take a first guess r; either $r^2 > x$, $r^2 < x$ or $r^2 = x$

- If $r^2 = x$, we can stop
- If $r^2 > x$, then $r > \sqrt{x}$, and $x/r < \sqrt{x}$
- If $r^2 < x$, then $r < \sqrt{x}$, and $x/r > \sqrt{x}$

Therefore, in the latter two cases, we can replace r with average of r and x/r, and try again