Data Frames and Control

36-350

3 September 2014

Agenda

- 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
  `v[5]`, ‘v[“name”]
- Arrays: multi-dimensional generalization of vectors `a[5,6,2]`, `a[,6,]`, `a[rowname, colname, layername]`
- Matrices: special 2D arrays with matrix math
  `m[5,6]`, `m[,6]`, `m[,colname]`
- Lists: series of values of mixed types
  `l[[3]]`, `l$name`
- Dataframes: hybrid of matrix and list

Dataframes, Encore

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

Creating an example dataframe

```r
library(datasets)
states <- data.frame(state.x77, abb=state.abb, region=state.region, division=state.division)
```

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

Column names are preserved or guessed if not explicitly set

===

```r
colnames(states)
```

```
##  [1] "Population" "Income"     "Illiteracy" "Life.Exp"   "Murder"    
##  [6] "HS.Grad"    "Frost"      "Area"       "abb"        "region"    
## [11] "division"
```

```r
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
##         abb region           division
## Alabama  AL  South East South Central
```

Dataframe access

- By row and column index

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

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

- By row and column names

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

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

Dataframe access (cont'd)

- All of a row:

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

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

Exercise: what class is `states["Wisconsin",]`?

Dataframe access (cont'd.)

- All of a column:

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

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

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

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

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

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

Dataframe access (cont'd.)

- Rows matching a condition:

```r
states[states$division=="New England", "Illiteracy"]
```

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

```r
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 dataframe can be assigned to:

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

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##    37.8    48.0    53.2    53.1    59.2    67.3
```

```r
states$HS.Grad <- states$HS.Grad/100
summary(states$HS.Grad)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   0.378   0.480   0.532   0.531   0.592   0.673
```

```r
states$HS.Grad <- 100*states$HS.Grad
```

with()

What percentage of literate adults graduated HS?

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

```
## [1] 42.19 67.72 59.16 40.67 63.30 64.35
```

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

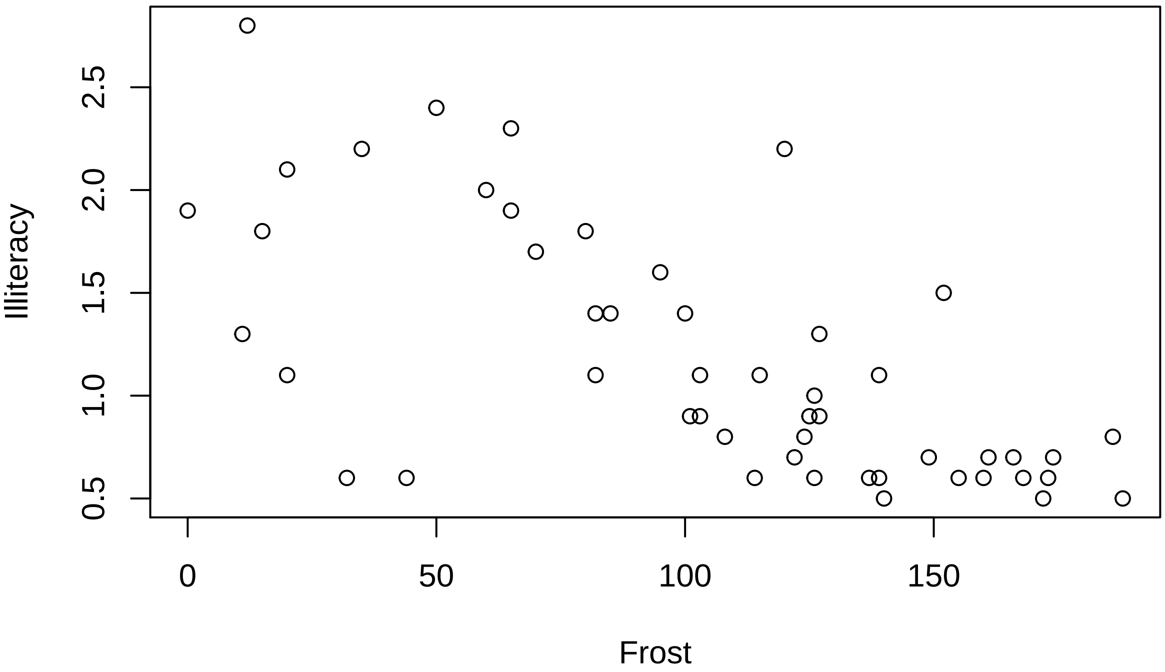
```r
with(states, head(100*(HS.Grad/(100-Illiteracy))))
```

```
## [1] 42.19 67.72 59.16 40.67 63.30 64.35
```

Data arguments

Lots of functions take `data` arguments, and look variables up in that data frame:

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



$R^2 = 0.45$, $p \approx 10^{-7}$

Conditionals

Have the computer decide what to do next - Mathematically:

$$|x| = \left\{ \begin{array}{cl} x & \mathrm{if}\ x \geq 0 \\ -x & \mathrm{if}\ x < 0 \end{array} \right. , \ \psi(x) = \left\{ \begin{array}{cl} x^2 & \mathrm{if}\ |x| \leq 1 \\ 2|x| - 1 & \mathrm{if}\ |x| > 1 \end{array} \right.$$

Exercise: plot ψ in R - Computationally:

```
if the country code is not "US", multiply prices by current exchange rate
```

if()

Simplest conditional:

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

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

`else` clause is optional

one-line actions don't need braces

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

Nested if()

`if` can *nest* arbitrarily deeply:

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

Can get ugly though

Combining Booleans: && and ||

`&` work `|` like `+` or `*`: combine terms element-wise

Flow control wants *one* Boolean value, and to skip calculating what's not needed

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

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

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

This *never* evaluates the complex expression on the right

Use `&&` and `||` for control, `&` and `|` for subsetting

Iteration

Repeat similar actions multiple times:

```
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] 0.0000 0.6931 1.0986 1.3863 1.6094 1.7918 1.9459
```

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)`) and **loops through** the **body* until it runs through the vector

"**iterates over** the vector"

N.B., there is a better way to do this job!

The body of the for() loop

Can contain just about anything, including: - if() clauses - 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 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() vs. 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

Avoiding iteration

R has many ways of *avoiding* iteration, by acting on whole objects - It's conceptually clearer - It leads to simpler code - It's 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
```

or a triple `for()` loop for matrix multiplication vs. `a %*% b`

Advantages of vectorizing

- Clarity: the syntax is about *what* we're doing
- Concision: we write less
- Abstraction: the syntax hides *how the computer does it*
- Generality: same syntax works for numbers, vectors, arrays, ... - Speed: modifying big vectors over and over is slow in R; work gets done by optimized low-level code

Vectorized calculations

Many functions are set up to vectorize automatically

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

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

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

```
## [1] 0.0000 0.6931 1.0986 1.3863 1.6094 1.7918 1.9459
```

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)
```

1st argument is a Boolean vector, then pick from the 2nd or 3rd vector arguments as `TRUE` or `FALSE`

Summary

- Dataframes
- `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`; the strings "TRUE" and "FALSE" count as you'd hope; most everything else gives an error

Advice: Don't play games here; try to make sure control expressions are getting Boolean values

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

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

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

switch()

Simplify 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 (off-line)

Set `type.of.summary` to, succesively, "mean", "median", "histogram", and "mode", and explain what happens

Unconditional iteration

```
repeat {
  print("Help! I am Dr. Morris Culpepper, trapped in an endless loop!")
}
```

"Manual" control over iteration

```
repeat {
  if (watched) { next() }
  print("Help! I am Dr. Morris Culpepper, trapped in an endless loop!")
  if (rescued) { break() }
}
```

`break()` exits the loop; `next()` skips the rest of the body and goes back into the loop

both work with `for()` and `while()` as well

Exercise: how would you replace `while()` with `repeat()`?

Babylonian Method of Root Finding

(Often attributed to Heron of Alexandria, about 2000 yrs later)

Given: x, find $\sqrt{x}$

Take a first guess r; either $r^2 > x$, $r^2 < x$ or $r^2 = x$

If $r^2 = x$, stop

If $r^2 > x$, then $r > \sqrt{x}$, but $x/r < x/\sqrt{x} = \sqrt{x}$
If $r^2 < x$, then $x/r > \sqrt{x}$

$\therefore$ Replace r with average of r and x/r, and try again