



Lecture 3: Data Frames and Control

S. Morteza Najibi

February 27, 2019

- ▶ Making and working with data frames
- ▶ Conditionals: switching between different calculations
- ▶ Iteration: Doing something over and over
- ▶ Vectorizing: Avoiding explicit iteration

- ▶ 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

- ▶ 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

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

`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

```
colnames(states)
```

```
[1] "Population" "Income"      "Illiteracy" "Life.Exp"    "Murder"
[6] "HS.Grad"    "Frost"       "Area"       "abb"         "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
      abb region          division
Alabama  AL   South East South Central
```

▶ By row and column index

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

```
[1] 0.7
```

▶ By row and column names

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

```
[1] 0.7
```

▶ 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
abb		region		division				
Wisconsin	WI	North Central	East	North Central				

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

▶ 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
```

▶ 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
```

Parts or all of the dataframe 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
```

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
```

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

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

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

Have the computer decide what to do next - Mathematically:

$$|x| = \begin{cases} x & \text{if } x \geq 0 \\ -x & \text{if } x < 0 \end{cases}, \quad \psi(x) = \begin{cases} x^2 & \text{if } |x| \leq 1 \\ 2|x| - 1 & \text{if } |x| > 1 \end{cases}$$

Exercise: plot ψ in R - Computationally:

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

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
```

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

& 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

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] 0.0000000 0.6931472 1.0986123 1.3862944 1.6094379 1.7917595 1.9459101
```

```
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!

Can contain just about anything, including: - if() clauses - other for() loops (nested iteration)

```
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 (max(x) - 1 > 1e-06) {  
  x <- sqrt(x)  
}
```

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() 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

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)

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`

- ▶ 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

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

```
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

- ▶ Dataframes
- ▶ `if`, nested `if`, `switch`
- ▶ Iteration: `for`, `while`
- ▶ Avoiding iteration with whole-object (“vectorized”) operations

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
```

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")
```

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


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

```
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()`?