Statistical Computing (36-350)

Lecture 3: Flow Control

Cosma Shalizi

4 September 2013

Agenda

- Conditionals: Switching between doing different things
- Iteration: Doing similar things many times
- Vectorizing: Avoiding explicit iteration

ABSOLUTELY ESSENTIAL READING FOR FRIDAY: Sec. 7.1 of the textbook
MERELY USEFUL READING: Chapters 3–5, "extended example" sections optional

Conditionals

Have the computer decide which calculation to do, based on the data
Mathematically:

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

or

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

(EXERCISE: plot $\psi(x)$ in R)
or

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

Simplest conditional: `if`

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

The condition in `if` needs to give *one* TRUE or FALSE value
`else` clause is optional
EXERCISE: What if `x` is a numeric vector?

What Is Truth?

Any valid numerical value except 0 counts as TRUE; 0 is FALSE; most non-numerical values choke:

```
> if(1) {"Truth!"} else {"Falsehood!"}
[1] "Truth!"
> if(-1) {"Truth!"} else {"Falsehood!"}
[1] "Truth!"
> if(0) {"Truth!"} else {"Falsehood!"}
[1] "Falsehood!"
> if("TRUE") {"Truth!"} else {"Falsehood!"}
[1] "Truth!"
> if("TRUTH") {"Truth!"} else {"Falsehood!"}
Error in if ("TRUTH") { : argument is not interpretable as logical
> if("c") {"Truth!"} else {"Falsehood!"}
Error in if ("c") { : argument is not interpretable as logical
> if(NULL) {"Truth!"} else {"Falsehood!"}
Error in if (NULL) { : argument is of length zero
> if(NA) {"Truth!"} else {"Falsehood!"}
Error in if (NA) { : missing value where TRUE/FALSE needed
> if(NaN) {"Truth!"} else {"Falsehood!"}
Error in if (NaN) { : argument is not interpretable as logical
```

Single or Double?

Boolean operators & and | are like arithmetic operators: act elementwise on vectors, every term evaluated

```
> c(TRUE,TRUE) & c(TRUE,FALSE)
[1]  TRUE FALSE
```

Flow control wants: single Boolean values, don't calculate what we don't need

Solution: && and ||
Go left to right, stop when answer is fixed

```
> (0>0) & ("c"+1)
Error in "c" + 1 : non-numeric argument to binary operator
> (0>0) && ("c"+1)
[1] FALSE
```

Now imagine some complicated calculation for the second term; R skips it because it doesn't matter!

Applied to vectors, the double-Booleans take the first element of each

```
> c(FALSE,FALSE) | c(TRUE,FALSE)
[1]  TRUE FALSE
> c(FALSE,FALSE) || c(TRUE,FALSE)
[1] TRUE
> c(FALSE,FALSE) || c(FALSE,TRUE)
[1] FALSE
```

Generally: Use && and || for flow control, try not to give them vector arguments

Nested ifs

Conditionals can **nest** arbitrarily deeply:

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

switch

Nesting `if/else` clauses always works, but gets complex
Simplify with `switch`: give a variable to select on, and then values for each option:

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

EXERCISE: Set `x <- c(5,7,8)` and run this with `type.of.summary` set to, successively, `"mean"`, `"median"`, `"histogram"` and `"mode"`.

Iteration: Doing Similar Things Multiple Times

Repeat the same, or a very similar, action a certain number of times:

```
n <- 10
table.of.logarithms <- vector(length=n)
table.of.logarithms
for (i in 1:n) {
  table.of.logarithms[i] <- log(i)
}
table.of.logarithms
```

`for` increments a **counter** (here `i`) along a vector (here `1:n`), and **loops through** the **body** until it runs through the vector
N.B., there is a better way to do this particular job

Combining for and if

```
x <- c(-5,7,-8,0)
y <- vector(length=length(x))
for (i in 1:length(x)) {
  if (x[i] >= 0) {
    y[i] <- x[i]
  } else {
    y[i] <- -x[i]
  }
}
y # now c(5,7,8,0)
```

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

while: Conditional Iteration

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

Condition in the argument to `while` must be a single TRUE/FALSE value, as with `if`

Loop is executed over and over until the condition is FALSE

⇒ goes forever if the condition is always TRUE

⇒ never begins unless the condition starts as TRUE

EXERCISE: How would you replace a `for` loop with a `while` loop?

Unconditional iteration

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

More useful:

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

Always enters the loop at least once, even if `rescued` is true

`break()` exits the loop; `next()` skips the rest of the body and goes back into the loop (both work on `for` and `while` too)

EXERCISE: How would you replace `while` with `repeat`?

Allocating Resources by Random Tinkering

Recall: our linear factory makes cars and trucks from labor and steel
Available resources (1600 hours, 70 tons) are completely employed by making 10 cars and 20 trucks
Exactly solved by linear algebra
Suppose didn't know linear algebra, and we didn't care if we used *all* the resources so long as the slack wasn't very large
Find solution by starting with an arbitrary plan and tinkering with it until it meets constraints

```
factory <- matrix(c(40,1,60,3),nrow=2,
  dimnames=list(c("labor","steel"),c("cars","trucks")))
available <- c(1600,70); names(available) <- rownames(factory)
slack <- c(8,1); names(slack) <- rownames(factory)
output <-c(30,20); names(output) <- colnames(factory)
```

How it works

```
passes <- 0 # How many times have we  been around the loop?
repeat {
   passes <- passes + 1
   needed <- factory %*% output # What do we need for that output level?
   if (all(needed <= available) && all((available - needed) <= slack)) {
     break() # If we're not using too much and inside the slack, we're done
   }
   if (all(needed > available)) {   # If we're using too much of everything
     output <- output * 0.9         # cut back by 10%
     next()
   }
   if (all(needed < available)) {   # If we're using too little of everything
     output <- output * 1.1         # increase by 10%
     next()
   }
   # If we're using too much of some resources but not others, randomly
   # tweak the plan by up to 10%
   output <- output * (1+runif(length(output),min=-0.1,max=0.1))
}
```

Typical output, after starting from 30 cars and 20 trucks:

```
> round(output,1)
  cars trucks
  10.4   19.7
> round(needed,1)
       [,1]
labor 1596.1
steel   69.4
> passes
[1] 3452
```

i.e., it adjusted the plan 3452 times
vs. 10 cars, 20 trucks at full utilization
Homework will examine and improve this

Avoiding Iteration

R gives a lot of ways to *avoid* iteration, by acting on whole objects

- Conceptually clearer
- Simpler code
- Faster (sometimes a little, sometimes drastically)

Lots of these are about **vectorizing** calculations

We have already seen this!

How many programming languages add the vectors `a` and `b`:

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

How R adds the vectors `a` and `b`:

```
c <- a+b
```

Advantages:

- Clarity: the syntax is about *what* we are doing
- Abstraction: the syntax hides *how the computer does it*
  The same syntax works for numbers, vectors, matrices, 13-dimensional arrays
- Concision: one line vs. four
- Speed: modifying big vectors over and over is slow in R, this gets passed off to optimized low-level code (usually the least important advantage)

Disadvantages:

- You have to learn to think about whole objects, not just parts
- Code tends to not look very intimidating

Vectorized Calculations

Many functions are set up to vectorize automatically

```
abs(x) # Absolute value of each element in x
log(x) # Logarithm of each element in x
```

Conditionality with `ifelse()`:

```
ifelse(x<0,-x,x) # Pretty much the same as abs(x)
ifelse(x^2>1,2*abs(x)-1,x^2)
```

First argument a Boolean vector, then pick from the second or third arguments as TRUE or FALSE
See also `apply()` from last time
Will come back to this in great detail later

Repeating

`rep(x,n)`: Repeat `x`, `n` times
`seq()`: Produce a sequence; useful, flexible, see textbook, sec. 2.4.4, and recipe 2.7 in the cookbook

Arrays with Repeated Structure

All combinations of values from vectors: `expand.grid`

```
> expand.grid(v1=c("lions","tigers"),v2=c(0.1,1.1))
      v1  v2
1  lions 0.1
2 tigers 0.1
3  lions 1.1
4 tigers 1.1
```

Makes a data frame so can combine different types
More than two input vectors is fine

Arrays with Repeated Structure (cont'd.)

Combinations of inputs to a function: `outer`

```
> outer(c(1,3,5),c(2,3,7),'*')
     [,1] [,2] [,3]
[1,]    2    3    7
[2,]    6    9   21
[3,]   10   15   35
```

N.B.: Special quotation marks for multiplication sign; similarly for other operators

This one gets its own abbreviated operator:

```
c(1,3,5) %o% c(2,3,7)
```

Any two-argument vectorized function works:

```
> outer(c(1024,1000),c(2,10),log)
          [,1]   [,2]
[1,] 10.000000 3.0103
[2,]  9.965784 3.0000
```

(What is the second argument of `log`?)

replicate(): Do the *exact* same thing many times

Why would we ever want to do that? When our code is somewhat random

```
# Take a sample of size 1000 from the standard exponential
rexp(1000,rate=1)
# Take the mean of such a sample
mean(rexp(1000,rate=1))
# Draw 1000 such samples, and take the mean of each one
replicate(1000,mean(rexp(1000),rate=1))
# Plot the histogram of sample means
hist(replicate(1000,mean(rexp(1000,rate=1))))
```

```
# Equivalent to that last, but dumb
sample.means <- vector(length=1000)
for (i in 1:length(sample.means)) {
   sample.means[i] <- mean(rexp(1000,rate=1))
}
hist(sample.means)
```

Summary

1. Conditions: Use `if ...else` and `switch()` to let the data pick different calculations or actions
2. Iteration: Use `for()`, `while()` and `repeat()` to do similar things a certain number of times, or while conditions hold, or until conditions are met
3. Vectorizing: Explicit iteration is often unclear, slow, and needlessly detailed; avoid it by working with whole objects