Statistical Computing (36-350)

Lecture 2: More Data Structures

Cosma Shalizi

29 August 2012

Admin Stuff

Homework 1: Out today
Office hours: Wednesdays, 11:30–1:30, Baker Hall 229C, or by appointment

Agenda

- Matrices
- Lists
- Data frames
- Structures of structures

Recall from last time

Data has different types or classes
Data structures group related values
Vectors are the basic data structure: sequence of values of the same type

```
> x <- c(8, 7, 10, 45)
> x[1]
[1] 8
> x[-4]
[1] 8 7 10
> x[x>9]
[1] 10 45
```

Vector structures are vectors + extra attributes
Multi-dimensional arrays are vector structures: access through the fancy multiple indices, or through the underlying vector

```
> x.arr <- array(x,dim=c(2,2))
> x.arr
     [,1] [,2]
[1,]    8   10
[2,]    7   45
> x.arr[2,2]
[1] 45
> x.arr[4]
[1] 45
```

Running example: resource allocation (continued)

Factory can make cars or trucks
Each car takes 40 hours of labor and 1 ton of steel
Each truck takes 60 hours of labor and 3 tons of steel
1600 hours and 70 tons per week
How many cars and trucks can it make each week?
Can it make (say) 20 trucks and 10 cars per week?
Motivation: step towards linear (mathematical) programming and optimization

Matrices

In R, a matrix is a specialization of a 2D array

```
> factory <- matrix(c(40,1,60,3),nrow=2)
> factory
     [,1] [,2]
[1,]   40   60
[2,]    1    3
> is.array(factory)
[1] TRUE
> is.matrix(factory)
[1] TRUE
```

could also specify `ncol`, and/or `byrow=TRUE` to fill by rows.
Element-wise operations with the usual arithmetic and comparison operators (e.g., `factory/3`)
Compare whole matrices with `identical()` or `all.equal()`

Matrix multiplication has a special operator:

```
> six.sevens <- matrix(rep(7,6),ncol=3)
> six.sevens
     [,1] [,2] [,3]
[1,]    7    7    7
[2,]    7    7    7
> factory %*% six.sevens # [2x2] * [2x3]
     [,1] [,2] [,3]
[1,]  700  700  700
[2,]   28   28   28
> six.sevens %*% factory # [2x3] * [2x2]
Error in six.sevens %*% factory : non-conformable arguments
```

Multiplying by a vector:

```
> output <- c(10,20)
> factory %*% output
     [,1]
[1,] 1600
[2,]   70
> output %*% factory
     [,1] [,2]
[1,]  420  660
```

R silently casts the vector as a row or column matrix

Matrix transpose:

```
> t(factory)
     [,1] [,2]
[1,]   40    1
[2,]   60    3
```

Matrix determinant:

```
> det(factory)
[1] 60
```

Extracting or replacing the diagonal:

```
> diag(factory) # What's the diagonal of the matrix?
[1] 40  3
> diag(factory) <- c(35,4) # Change it
> factory # See that it changed
     [,1] [,2]
[1,]   35   60
[2,]    1    4
> diag(factory) <- c(40,3) # Set it back for later
```

Creating a diagonal matrix or an identity matrix:

```
> diag(c(3,4))
     [,1] [,2]
[1,]    3    0
[2,]    0    4
> diag(2)
     [,1] [,2]
[1,]    1    0
[2,]    0    1
```

Inverting a matrix:

```
> solve(factory)
            [,1]       [,2]
[1,]  0.05000000 -1.0000000
[2,] -0.01666667  0.6666667
> factory %*% solve(factory) # Check that this does what I claim
     [,1] [,2]
[1,]    1    0
[2,]    0    1
```

Why is it called `solve()`?

Solving the linear system $\mathbf{A}\vec{x} = \vec{b}$, for unknown vector $\vec{x}$:

```
> available <- c(1600,70)
> solve(factory,available)
[1] 10 20
> factory %*% solve(factory,available)
     [,1]
[1,] 1600
[2,]   70
```

Names in Matrices

We can name either rows or columns or both, with `rownames()` and `colnames()`

These are just character vectors, and we use the same function to get and to set their values

Names help us understand what we're working with

Names can be used to coordinate different objects

Example: someone gets the order of cars and trucks wrong

```
> rownames(factory) <- c("labor","steel")
> colnames(factory) <- c("cars","trucks")
> factory
      cars trucks
labor   40     60
steel    1      3
> output <- c(20,10)
> names(output) <- c("trucks","cars")
> available <- c(1600,70)
> names(available) <- c("labor","steel")
> factory %*% output # But we've got cars and trucks mixed up!
      [,1]
labor 1400
steel   50
> factory %*% output[colnames(factory)]
      [,1]
labor 1600
steel   70
> all(factory %*% output[colnames(factory)] <= available[rownames(factory)])
[1] TRUE
```

Notice: Last lines don't have to change if we add motorcycles as output or rubber and glass as inputs (abstraction again)

Doing the same thing to each row or column

Take the mean: `rowMeans()`, `colMeans()`: input is matrix, output is vector. Also `rowSums()`, etc.
`summary()`: Applies vector-style summary to each column
`apply()`, takes 3 arguments: matrix, then 1 for rows and 2 for columns, the name of the function to apply to each

```
> rowMeans(a.matrix)
[1] 22.5  6.0
> apply(a.matrix,1,mean)
[1] 22.5  6.0
```

Lists

Sequence of values, *not* necessarily all of the same type

```
> my.distribution <- list("exponential",7,FALSE)
> my.distribution
[[1]]
[1] "exponential"

[[2]]
[1] 7

[[3]]
[1] FALSE
```

Most of things which you can do with vectors you can also do with lists

Accessing bits of lists

Access: can use [] as with vectors
or use [[]], but only with a single index (and it drops names and structures)

```
> is.character(my.distribution)
[1] FALSE
> is.character(my.distribution[[1]])
[1] TRUE
> my.distribution[2]^ 2
Error in my.distribution[2]^2 : non-numeric argument to binary operator
> my.distribution[[2]]^2
[1] 49
```

(What happens when you try [[]] on a vector?)

Add to lists with `c()` (also works with vectors):

```
> my.distribution <- c(my.distribution,7)
> my.distribution
[[1]]
[1] "exponential"
[[2]]
[1] 7
[[3]]
[1] FALSE
[[4]]
[1] 7
```

Chop off the end of a list by setting length to something smaller (also works with vectors):

```
> length(my.distribution)
[1] 4
> length(my.distribution) <- 3
> my.distribution
[[1]]
[1] "exponential"
[[2]]
[1] 7
[[3]]
[1] FALSE
```

Names in lists

We can name some or all of the elements of a list

```
> names(my.distribution) <- c("family","mean","is.symmetric")
> my.distribution
$family
[1] "exponential"
$mean
[1] 7
$is.symmetric
[1] FALSE
```

Then we access by name, using $ (which removes names and structure):

```
> my.distribution$family
[1] "exponential"
> my.distribution[["family"]]
[1] "exponential"
> my.distribution["family"]
$family
[1] "exponential"
```

Using names when we make the list:

```
> another.distribution <- list(family="gaussian",mean=7,sd=1,is.symmetric=TRUE)
> another.distribution
$family
[1] "gaussian"
$mean
[1] 7
$sd
[1] 1
$is.symmetric
[1] TRUE
```

or after:

```
> my.distribution$was.estimated <- FALSE
> my.distribution[["last.updated"]] <- "2011-08-30"
> my.distribution
$family
[1] "exponential"
$mean
[1] 7
$is.symmetric
[1] FALSE
$was.estimated
[1] FALSE
$last.updated
[1] "2011-08-30"
```

Remove an entry in the list by assigning it the value `NULL`; try `my.distribution$was.estimated<-NULL`

Key-Value Pairs

Lists give us a way to store and look data up by name rather than number (**key-value pairs**, **dictionary**, **associative array**, **hash**)
If all our distributions have a component named `family`, we can look it up by name without caring where it is in the list
(More abstraction)

Data frames

Data frame = classic data table, with n rows for cases, and p columns for variables

Lots of the really-statistical parts of R presume data frames

Not just a matrix because every column can be of a different type

A hybrid of a matrix and a list; can access columns either like a matrix or like named parts of a list

Many functions for matrices also work on data frames (`rowSums()`, `summary()`, `apply()`, `...`)

Cannot do matrix multiplication on a data frame even if it's all numbers

Example:

```
> a.matrix <- matrix(c(35,8,10,4),nrow=2)
> colnames(a.matrix) <- c("v1","v2")
> a.matrix
     v1 v2
[1,] 35 10
[2,]  8  4
> a.matrix$v1 # The $ access operator doesn't work on a matrix
Error in a.matrix$v1 : $ operator is invalid for atomic vectors
> a.data.frame <- data.frame(a.matrix,logicals=c(TRUE,FALSE))
> a.data.frame
  v1 v2 logicals
1 35 10     TRUE
2  8  4    FALSE
> a.data.frame$v1 # But $ does work on a data frame
[1] 35  8
> a.data.frame[,"v1"]
[1] 35  8
> a.data.frame[1,]
  v1 v2 logicals
1 35 10     TRUE
> colMeans(a.data.frame)
      v1       v2 logicals
    21.5      7.0      0.5
```

We can add rows or columns to an array or data-frame with `rbind()` and `cbind()`, but be careful about forced type conversions

```
> rbind(a.data.frame,list(v1=-3,v2=-5,logicals=TRUE))
  v1 v2 logicals
1 35 10     TRUE
2  8  4    FALSE
3 -3 -5     TRUE
> rbind(a.data.frame,c(3,4,6))
  v1 v2 logicals
1 35 10        1
2  8  4        0
3  3  4        6
```

Structures of structures

Lists of lists, lists of vectors, lists of lists of lists of vectors...
This **recursion** lets us build arbitrarily complicated data structures from the basic ones
Lots of complicated objects are lists of data structures

Example: Eigenstuff

`eigen()` finds eigenvalues and eigenvectors of a matrix
return value is a list of a vector (of eigenvalues) and a matrix (of eigenvectors)

```
> eigen(factory)
$values
[1] 41.556171  1.443829

$vectors
           [,1]       [,2]
[1,] 0.99966383 -0.8412758
[2,] 0.02592747  0.5406062

> class(eigen(factory))
[1] "list"
> str(eigen(factory))
List of 2
 $ values : num [1:2] 41.56 1.44
 $ vectors: num [1:2, 1:2] 0.9997 0.0259 -0.8413 0.5406
```

With complicated objects, you can access parts of parts (of parts...):

```
> factory %*% eigen(factory)$vectors[,2]
           [,1]
[1,] -1.2146583
[2,]  0.7805429
> eigen(factory)$values[2] * eigen(factory)$vectors[,2]
[1] -1.2146583  0.7805429
> eigen(factory)$values[2]
[1] 1.443829
> eigen(factory)[[1]][[2]] # NOT [[1, 2]]
[1] 1.443829
```

Summary

- Matrices act like you'd hope they would
- Lists let us combine different types of data
- Data frames are hybrids of matrices and lists, for classic tabular data
- Use names of components to make data more meaningful and control access
- Recursion lets us build complicated structures