More Data Structures

IS 665
Yegin Genc

Agenda

- Matrices
- Arrays
- Lists
- Dataframes
- Structures of structures

Vector structures, starting with arrays

Many data structures in R are made by adding bells and whistles to vectors, so “vector structures”

A **matrix** in R is a collections of homogeneous elements arranged in 2 dimensions

```
matrix(1:15, nrow = 4)
```

```
     [,1] [,2] [,3] [,4]
[1,]    1    5    9   13
[2,]    2    6   10   14
[3,]    3    7   11   15
[4,]    4    8   12    1
```

Matrices

- A matrix is a vector with a dim attribute, i.e. an integer vector giving the number or rows and columns
- To create matrices us matrix()
- The functions dim(), nrow() and ncol() provide the attributes of the matrix
- Rows and columns can have names, dimnames(), rownames(), colnames()

Matrices

```
factory <- matrix(c(40,1,60,3),nrow=2 )
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

Gets a special operator

```r
six.sevens <- matrix(rep(7,6),ncol=3)
six.sevens
```

```
     [,1] [,2] [,3]
[1,]    7    7    7
[2,]    7    7    7
```

```r
factory %*% six.sevens # [2x2] * [2x3]
```

```
     [,1] [,2] [,3]
[1,]  700  700  700
[2,]   28   28   28
```

What happens if you try `six.sevens %*% factory`?

Matrix operators

Transpose:

```
t(factory)
```

```
     [,1] [,2]
[1,]   40    1
[2,]   60    3
```

Determinant:

```
det(factory)
```

```
[1] 60
```

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

```r
rownames(factory) <- c("labor","steel")
colnames(factory) <- c("cars","trucks")
factory
```

```
      cars trucks
labor   40     60
steel    1      3
```

```r
available <- c(1600,70)
names(available) <- c("labor","steel")
```

Doing the same thing to each row or column

Take the mean: rowMeans(), colMeans(): input is matrix, output is vector. Also rowSums(), etc.

summary(): vector-style summary of column

```
colMeans(factory)
```

```
 cars trucks
 20.5   31.5
```

```
summary(factory)
```

```
      cars           trucks
 Min.   : 1.00   Min.   : 3.00
 1st Qu.:10.75   1st Qu.:17.25
 Median :20.50   Median :31.50
 Mean   :20.50   Mean   :31.50
 3rd Qu.:30.25   3rd Qu.:45.75
 Max.   :40.00   Max.   :60.00
```

Extra

apply(), takes 3 arguments: the array or matrix, then 1 for rows and 2 for columns, then name of the function to apply to each

```
rowMeans(factory)
```

```
labor steel
   50     2
```

```
apply(factory,1,mean)
```

```
labor steel
   50     2
```

What would `apply(factory,1,sd)` do?

Arrays

arrays are basically matrices in higher dimensions

```r
x <- c(7, 8, 10, 45 , 70, 80 , 100, 250)
x.arr <- array(x,dim=c(2,2,2))
x.arr
```

```
, , 1

     [,1] [,2]
[1,]    7   10
[2,]    8   45

, , 2

     [,1] [,2]
[1,]   70  100
[2,]   80  250
```

dim says how many rows and columns; filled by columns

Can have $3, 4, \ldots n$ dimensional arrays; dim is a length-n vector

Arrays cntd.

Some properties of the array:

```
dim(x.arr)
```

```
[1] 2 2 2
```

```
is.vector(x.arr)
```

```
[1] FALSE
```

```
is.array(x.arr)
```

```
[1] TRUE
```

Arrays cntd.

```r
typeof(x.arr)
```

```
[1] "double"
```

```r
str(x.arr)
```

```
 num [1:2, 1:2, 1:2] 7 8 10 45 70 80 100 250
```

```r
attributes(x.arr)
```

```
$dim
[1] 2 2 2
```

typeof() returns the type of the *elements*

str() gives the **structure**: here, a numeric array, with three dimensions, both indexed 1–2, and then the actual numbers

Exercise: try all these with x

Accessing and operating on arrays

Can access a 2-D array either by pairs of indices or by the underlying vector:

```
x <- c(7, 8, 10, 45)
x.arr <- array(x,dim=c(2,2))
x.arr
```

```
     [,1] [,2]
[1,]    7   10
[2,]    8   45
```

```
x.arr[1,2]
```

```
[1] 10
```

```
x.arr[3]
```

```
[1] 10
```

Accessing and operating on arrays

Omitting an index means “all of it”:

```
x.arr[c(1:2),2]
```

```
[1] 10 45
```

```
x.arr[,2]
```

```
[1] 10 45
```

Functions on arrays

Using a vector-style function on a vector structure will go down to the underlying vector, *unless* the function is set up to handle arrays specially:

```
which(x.arr > 9)
```

```
[1] 3 4
```

Functions on arrays

Many functions *do* preserve array structure:

```
y <- -x
y.arr <- array(y,dim=c(2,2))
y.arr + x.arr
```

```
     [,1] [,2]
[1,]    0    0
[2,]    0    0
```

Others specifically act on each row or column of the array separately:

```
rowSums(x.arr)
```

```
[1] 17 53
```

We will see a lot more of this idea

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 what you can do with vectors you can also do with lists

Expanding and contracting lists

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 the 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
```

Extra - Accessing pieces of lists

Can use [ ] as with vectors

or use [[ ]], but only with a single index

[[ ]] drops names and structures, [ ] does not

```
is.character(my.distribution)
```

```
[1] FALSE
```

```
is.character(my.distribution[[1]])
```

```
[1] TRUE
```

```
my.distribution[[2]]^2
```

```
[1] 49
```

What happens if you try `my.distribution[2]^2`?

What happens if you try [[ ]] on a vector?

Dataframes

Dataframe = the classic data table, n rows for cases, p columns for variables

Not just a matrix because *columns can have different types*

Many matrix functions also work for dataframes (rowSums(), summary(), apply())

but no matrix multiplication of dataframes, even if all columns are numeric

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

```
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

```
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
```

Dataframe access

- By row and column index

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

```
[1] 0.7
```

- By row and column names

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

```
[1] 0.7
```

Dataframe access (cont'd)

- 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",]`?

Dataframe access (cont'd.)

- 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
```

Dataframe access (cont'd.)

- 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 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
```

Adding rows and columns

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

```
a.data.frame()
rbind(a.data.frame,list(v1=-3,v2=-5,logicals=TRUE))
rbind(a.data.frame,c(3,4,6))
```

*Internally, a dataframe is basically a list of vectors

Summary

- Matrices act like you'd hope they would
- Arrays add multi-dimensional structure to vectors
- Lists let us combine different types of data
- Dataframes are hybrids of matrices and lists, for classic tabular data

References

- http://www.stat.cmu.edu/~cshalizi/statcomp/
- https://www.r-project.org/
- https://www.rstudio.com/