Data Structure

Canhong Wen

2020-9-17

Agenda

- Vectors
- Arrays
- Matrices
- Lists
- Dataframes

Vectors

A **vector** is a sequence of values, all of the same type.

```
x <- c(7, 8, 10, 45)
x
```

```
## [1]  7  8 10 45
```

Another two ways to create vector:

```
y <- 1:10
y
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

```
seq(from=5, to=25, by=5)
```

```
## [1]  5 10 15 20 25
```

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

```
## [1] TRUE
```

`vector(length=6)` returns an empty vector of length 6; helpful for filling things up later.

```
weekly.hours <- vector(length=6, mode = "numeric")
weekly.hours
```

```
## [1] 0 0 0 0 0 0
```

```
weekly.hours[5] <- 8
weekly.hours
```

```
## [1] 0 0 0 0 8 0
```

Vector arithmetic

Operators apply to vectors "pairwise" or "elementwise":

```r
y <- c(-7, -8, -10, -45)
x+y
```

```
## [1] 0 0 0 0
```

```r
x*y
```

```
## [1]   -49   -64  -100 -2025
```

Recycling

Recycling repeat elements in shorter vector when combined with longer

```r
x + c(-7,-8)
```

```
## [1]  0  0  3 37
```

```r
x^c(1,0,-1,0.5)
```

```
## [1] 7.000000 1.000000 0.100000 6.708204
```

Single numbers are vectors of length 1 for purposes of recycling:

```r
2*x
```

```
## [1] 14 16 20 90
```

Can also do pairwise comparisons:

```r
x > 9
```

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

Note: returns Boolean vector

Boolean operators work elementwise:

```r
(x > 9) & (x < 20)
```

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

To compare whole vectors, best to use `identical()` or `all.equal()`:

```r
x == -y
```

```
## [1] TRUE TRUE TRUE TRUE
```

```r
identical(x,-y)
```

```
## [1] TRUE
```

```r
identical(c(0.5-0.3,0.3-0.1),c(0.3-0.1,0.5-0.3))
```

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

```r
all.equal(c(0.5-0.3,0.3-0.1),c(0.3-0.1,0.5-0.3))
```

```
## [1] TRUE
```

Functions on vectors

Lots of functions take vectors as arguments:

- `mean()`, `median()`, `sd()`, `var()`, `max()`, `min()`, `length()`, `sum()`: return single numbers
- `sort()` returns a new vector
- `hist()` takes a vector of numbers and produces a histogram, a highly structured object, with the side-effect of making a plot
- Similarly `ecdf()` produces a cumulative-density-function object
- `summary()` gives a five-number summary of numerical vectors
- `any()` and `all()` are useful on Boolean vectors

Addressing vectors

Vector of indices:

```
x[c(2,4)]
```

```
## [1]  8 45
```

Vector of negative indices

```
x[c(-1,-3)]
```

```
## [1]  8 45
```

(why that, and not `8 10`?)

Boolean vector:

```
x[x>9]
```

```
## [1] 10 45
```

```
y[x>9]
```

```
## [1] -10 -45
```

`which()` turns a Boolean vector in vector of TRUE indices:

```
places <- which(x > 9)
places
```

```
## [1] 3 4
```

```
y[places]
```

```
## [1] -10 -45
```

Named components

You can give names to elements or components of vectors

```
names(x) <- c("v1","v2","v3","fred")
names(x)
```

```
## [1] "v1"   "v2"   "v3"   "fred"
```

```
x[c("fred","v1")]
```

```
## fred   v1
##   45    7
```

note the labels in what R prints; not actually part of the value

`names(x)` is just another vector (of characters):

```
names(y) <- names(x)
sort(names(x))
```

```
## [1] "fred" "v1"   "v2"   "v3"
```

```
which(names(x)=="fred")
```

```
## [1] 4
```

Arrays

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

Most useful: **arrays**

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

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

`dim` says how many rows and columns; filled by columns

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

Some properties of the array:

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

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

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

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

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

```
## [1] TRUE
```

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

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

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

```
##  num [1:2, 1:2] 7 8 10 45
```

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

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

`typeof()` returns the type of the *elements*

`str()` gives the **structure**: here, a numeric array, with two 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.arr[1,2]
```

```
## [1] 10
```

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

```
## [1] 10
```

Omitting an index means "all of it":

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

```
## [1] 10 45
```

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

```
## [1] 10 45
```

Names in array

We can name any dimension, with `dimnames()`.

```
dimnames(x.arr)
```

```
## NULL
```

```
dimnames(x.arr)[[1]] <- c("r1","r2")
dimnames(x.arr)[[2]] <- c("v1","v2")
x.arr
```

```
##    v1 v2
## r1  7 10
## r2  8 45
```

```
dimnames(x.arr) # is a list
```

```
## [[1]]
## [1] "r1" "r2"
##
## [[2]]
## [1] "v1" "v2"
```

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

```
## v1 v2
##  7 10
```

```
x.arr["r1",]
```

```
## v1 v2
##  7 10
```

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
```

Many functions *do* preserve array structure:

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

```
##    v1 v2
## r1  0  0
## r2  0  0
```

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

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

```
## r1 r2
## 17 53
```

`apply()`, takes 3 arguments: the array, then the subscripts which the function will be applied over, e.g., 1 for rows and 2 for columns, then name of the function to apply to each.

```
rowMeans(x.arr)
```

```
##   r1   r2
##  8.5 26.5
```

```
apply(x.arr,1,mean)
```

```
##   r1   r2
##  8.5 26.5
```

Demo example: resource allocation

Factory makes cars and trucks, using labor and steel

- a car takes 40 hours of labor and 1 ton of steel
- a truck takes 60 hours and 3 tons of steel
- resources: 1600 hours of labor and 70 tons of steel each week

Matrices

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

```
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

```
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
```

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

Multiplying matrices and vectors

Numeric vectors can act like proper vectors:

```
output <- c(10,20)
factory %*% output
```

```
##      [,1]
## [1,] 1600
## [2,]   70
```

```
output %*% factory
```

```
##      [,1] [,2]
## [1,]  420  660
```

R silently casts the vector as either a row or a column matrix

Matrix operators

Transpose:

```
t(factory)
```

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

Determinant:

```
det(factory)
```

```
## [1] 60
```

The diagonal

The `diag()` function can extract the diagonal entries of a matrix:

```
diag(factory)
```

```
## [1] 40  3
```

It can also *change* the diagonal:

```
diag(factory) <- c(35,4)
factory
```

```
##      [,1] [,2]
## [1,]   35   60
## [2,]    1    4
```

Re-set it for later:

```
diag(factory) <- c(40,3)
```

Creating a diagonal or 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)
```

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

Why's it called "solve"" anyway?

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

```
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

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

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

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

```
output <- c(20,10)
names(output) <- c("trucks","cars")
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()`: 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
```

`apply()`, takes 3 arguments: the 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?

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

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?

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
```

Naming list elements

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
```

```
my.distribution[["family"]]
```

```
## [1] "exponential"
```

```
my.distribution["family"]
```

```
## $family
## [1] "exponential"
```

Lists have a special short-cut way of using names, `$` (which removes names and structures):

```
my.distribution[["family"]]
```

```
## [1] "exponential"
```

```
my.distribution$family
```

```
## [1] "exponential"
```

Names in lists

Creating a list with names:

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

Adding named elements:

```
my.distribution$was.estimated <- FALSE
my.distribution[["last.updated"]] <- "2019-08-30"
```

Removing a named list element, by assigning it the value `NULL`:

```
my.distribution$was.estimated <- NULL
```

Key-Value pairs

Lists give us a way to store and look up data by *name*, rather than by *position*

A really useful programming concept with many names: **key-value pairs**, **dictionaries**, **associative arrays**, **hashes**

If all our distributions have components named `family`, we can look that up by name, without caring where it is in the list

Dataframes

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

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

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

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

but no matrix multiplying dataframes, even if all columns are numeric

```
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"]  # Try a.matrix$v1 and see what happens
```

```
## [1] 35  8
```

```
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
```

```
## [1] 35  8
```

```
a.data.frame[,"v1"]
```

```
## [1] 35  8
```

```
colMeans(a.data.frame)
```

```
##       v1       v2 logicals
##     21.5      7.0      0.5
```

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

```
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
```

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

- 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
```

```
class(states["Wisconsin",])
```

```
## [1] "data.frame"
```

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

Dataframe access

- 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

- 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
```

Dataframe access

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
```

with()

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:

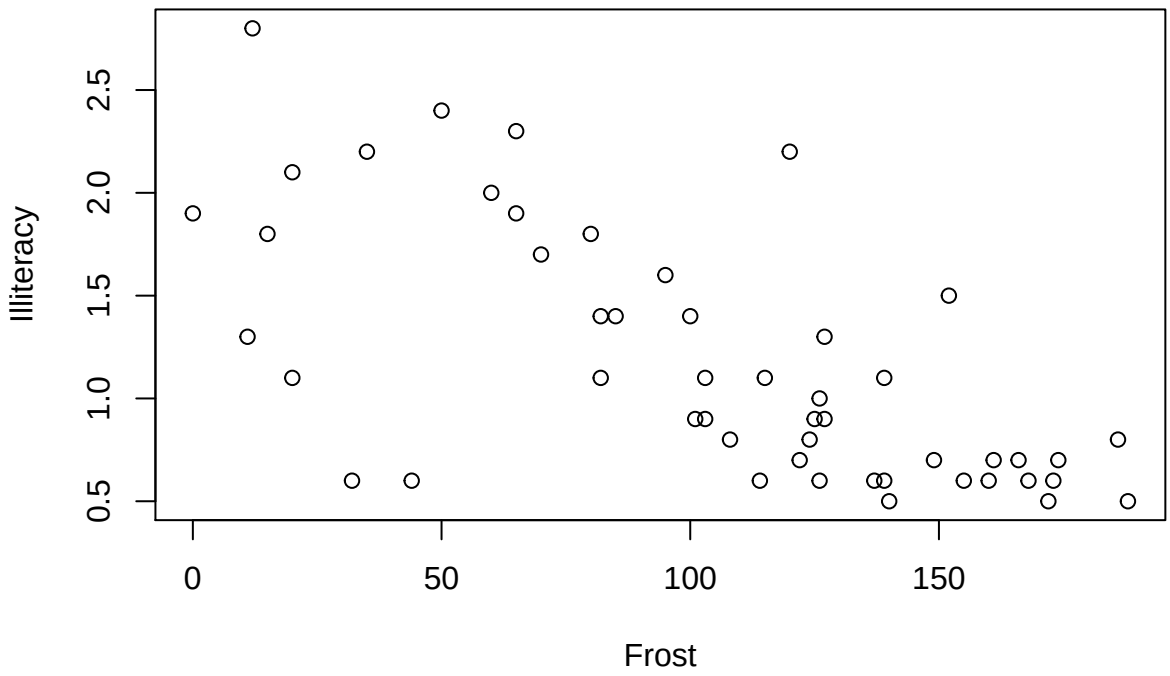
```
with(states, head(100*(HS.Grad/(100-Illiteracy))))
```

```
## [1] 42.18590 67.71574 59.16497 40.67278 63.29626 64.35045
```

Data arguments

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

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



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

Summary

- Data structure let us group related values together
- Vectors let us group values of the same type
- Arrays add multi-dimensional structure to vectors
- Matrices act like you'd hope they would
- Lists let us combine different types of data
- Dataframes are hybrids of matrices and lists, for classic tabular data