Stat 705
Class 2
Statistical computing with R

Richard P. Waterman

Wharton

Table of contents I

Today's module

Topics to be covered in this module:

- Last time
- Data structures
  1. Vectors
  2. Matrices
  3. Arrays
  4. Lists
- Subsetting, slicing and dicing
- Summary
- What you should be able to do now
- Next time

Last time

- R-studio and the project dialogue
- Accessing the help facility
- Arithmetic operators (+, -, *, / etc) and scientific functions (log, e, etc)
- Variable types: numeric, character, logical and factor
- Storing results in variables
- Logical and comparison operators

Vectors

Vectors store a sequence of values of the **same** type

```r
# A sequence of numerics
c(5.2,7.6,15.1)

## [1]  5.2  7.6 15.1

# A vector of  numerics stored into a variable
my.vec <- c(5.2,7.6,15.1)
# A vector of characters
my.sons <- c("Alex","Tom","David")
# A vector of logicals
logic.vec <- c(FALSE,TRUE,TRUE,FALSE)
```

Combining vectors

Use the "c" function to combine vectors. Separate the elements with commas.

```
my.vec1 <- c(5.2, 7.6, 15.1)  # A vector of  numerics
my.vec2 <- c(4, 21.2, 10) # A vector of  numerics
my.vec3 <- c(my.vec1, my.vec2) # The concatenation of the two vectors
print(my.vec3)

## [1]  5.2  7.6 15.1  4.0 21.2 10.0

### What if you try and combine vectors of different types?
my.sons <- c("Alex","Tom","David")  # A vector of characters
c(my.vec3,my.sons)    # R coerces the numbers to characters

## [1] "5.2"   "7.6"   "15.1"  "4"     "21.2"  "10"    "Alex"
## [8] "Tom"   "David"

# Note the quotes around the numbers
c(my.vec3,logic.vec)  # And coerces logicals to numerics

##  [1]  5.2  7.6 15.1  4.0 21.2 10.0  0.0  1.0  1.0  0.0
```

Easy vector creation

There are functions that make it easy to create vectors

```
a <- c(1:4)        # A consecutive sequence of integers. Use ":"
print(a)

## [1] 1 2 3 4

b <- seq(1,4,.5)  # A sequence with spacings of 0.5. The "seq" function.
print(b)

## [1] 1.0 1.5 2.0 2.5 3.0 3.5 4.0

# The "rep" function creates a sequence of replicates
d <- c(rep(10,5),rep(4,2))
print(d)

## [1] 10 10 10 10 10  4  4

# The "length" function tells you how many elements are in the vector
length(d)
```

Be very careful

- Notice how in the previous slide my variable names went a, b and d. I didn't use "c" as a variable name.
- That is because "c" is the built in function that combines its arguments (creates a vector) – don't mess with the letter "c".
- There is also another one letter function "t" which transposes a matrix.
- Advice: never use "c" and "t" as variable names.

Extracting and replacing elements in a vector

- The key operator is the "[" square brackets operator. It is used for extracting and replacing elements of a vector.
- You identify the elements of the vector by indicating their numerical index.
- In R, vectors start with the index 1. (In some languages they start at 0)

```
# letters is a built in character vector containing the lowercase letters
# Store them in a variable because I will mess with them
my.letters <- letters
my.letters

##  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n"
## [15] "o" "p" "q" "r" "s" "t" "u" "v" "w" "x" "y" "z"

length(my.letters)

## [1] 26
```

Extracting and replacing elements in a vector

```
my.letters[1:10]          # The first ten letters

##  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j"

my.letters[c(1,10,26)]    # The first, tenth and last letters

## [1] "a" "j" "z"

# You can also extract using a logical vector
my.letters[c(1:26) %%2 == 1] # Every other letter

##  [1] "a" "c" "e" "g" "i" "k" "m" "o" "q" "s" "u" "w" "y"
```

Extracting and replacing elements in a vector

```
my.letters[c(FALSE,TRUE,rep(FALSE,24))] # Extraction by logicals

## [1] "b"
```

Be very careful:

```
# When the extraction logical vector is not of length 26,
# R will automatically recycle it
# This can give lots of surprising results
my.letters[c(TRUE,FALSE)]

##  [1] "a" "c" "e" "g" "i" "k" "m" "o" "q" "s" "u" "w" "y"

# You can drop elements of a vector using the "-" prefix
new.letters <- my.letters[-c(1,2)] #I dropped the first two letters
print(new.letters)

##  [1] "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p"
## [15] "q" "r" "s" "t" "u" "v" "w" "x" "y" "z"
```

Extracting from a named vector

Extracting by name can be more robust if the data grows over time. You have to quote the name of the item you are extracting.

```
# A named vector of GPA's
gpas <- c("Math" = 3.4, "Verbal" = 3.7, "Analytics" = 3.9)

gpas["Verbal"]

## Verbal
##    3.7
```

Extracting and replacing elements in a vector

```
# Replacement of vector elements works the same way by using indexing
my.letters[1] <- "A"  # I replaced the first a with A
my.letters

##  [1] "A" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n"
## [15] "o" "p" "q" "r" "s" "t" "u" "v" "w" "x" "y" "z"

my.letters[c(4, 6, 1)] <- "Hello" # Replacing some arbitrary elements
my.letters

##  [1] "Hello" "b"     "c"     "Hello" "e"     "Hello" "g"
##  [8] "h"     "i"     "j"     "k"     "l"     "m"     "n"
## [15] "o"     "p"     "q"     "r"     "s"     "t"     "u"
## [22] "v"     "w"     "x"     "y"     "z"
```

In-class activity I

Problem synopsis:

You are a risk manager at an insurance company with major operations in Miami and Tokyo. The number of hurricanes hitting the main Japanese islands each year averages 3. The number of hurricanes hitting the Eastern US each year averages 1.75.

Objective:

You will be out of business if 8 or more hurricanes in total hit next year. Estimate the probability of this event.

Assumptions and parameters:

- Use independent Poisson random variables to estimate this probability.
- Use 10,000 yearly replicates in your simulation.

In-class activity I: process

1. Create a new .Rscript for this in-class activity
2. Then, sketch out your approach
3. Use the R help facility to learn how to create Poisson random variates (search using an uppercase P)
4. Set the random number seed and create variables for the parameters of the simulation
5. Generate 10,000 yearly events
6. Count how many of the yearly events have a total number of hurricanes ≥ 8

Discussion of your solution

Discussion.

Matrices

Matrices are a set of vectors, a vector of vectors. Here's a matrix:

$$\begin{pmatrix} 5 & 8 & 7 \\ 2 & 6 & 9 \\ 4 & 2 & 5 \\ 3 & 5 & 4 \end{pmatrix}$$

```
# Entering into R
my.matrix <- matrix( # The function is called "matrix"
                data = c(5,8,7,2,6,9,4,2,5,3,5,4),   #The raw data
                nrow = 4, # The number of rows
                ncol = 3) # The number of columns
print(my.matrix)

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

Matrices

Look carefully at the preceding slide. The matrix in R doesn't match the one I thought I was entering. That is because, by default, R will fill the matrix, by filling up the columns first. You can override this behaviour by using the byrow=TRUE argument.

```
my.matrix <- matrix( # The function is called "matrix"
                data = c(5,8,7,2,6,9,4,2,5,3,5,4),   #The raw data
                nrow = 4,    # The number of rows
                ncol = 3,    # The number of columns
                byrow=TRUE) # Filling up the matrix, one row at a time
print(my.matrix)

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

Now it matches!

Matrices

The functions `rownames` and `colnames` will allow you to set the row and column names of the matrix

```
colnames(my.matrix) <- c("Alpha", "Beta", "Gamma")
rownames(my.matrix) <- c("North","East","South", "West")
print(my.matrix)

##       Alpha Beta Gamma
## North     5    8     7
## East      2    6     9
## South     4    2     5
## West      3    5     4

colnames(my.matrix)

## [1] "Alpha" "Beta"  "Gamma"

rownames(my.matrix)

## [1] "North" "East"  "South" "West"
```

Extracting from a matrices

The extraction operator is still the square parenthesis "[". The first index is the row, the second the column.

```
my.matrix[4,2]            # Extract a single element

## [1] 5

my.matrix[,3]             # Extract a single column

## North  East South  West
##     7     9     5     4

my.matrix[2,]             # Extract a single row

## Alpha  Beta Gamma
##     2     6     9
```

Extracting from a matrices

Extract submatrices by indicating which rows and columns to keep.

```
my.matrix[,c(2,3)]                #Extract columns 2 and 3.

##       Beta Gamma
## North    8     7
## East     6     9
## South    2     5
## West     5     4

my.matrix[c(1,3),c(2,3)]          #Extract rows 1 and 3, columns 2 and 3.

##       Beta Gamma
## North    8     7
## South    2     5
```

Be very careful

Notice when we extracted the single column, it wasn't presented as a column of a matrix, but rather as a vector. R will automatically drop the dimension of a matrix if it doesn't need the additional dimension. The command to get the dimensions of a matrix is `dim`.

```
dim(my.matrix)

## [1] 4 3

dim(my.matrix[,3])       # Extract a single column and report dimensions

## NULL
```

The dimensions have disappeared because the single column has been dropped to a vector. You can stop this happening with the `drop = FALSE` argument to the extraction operator "[".

Be very careful

You can stop this happening with the drop = FALSE argument to the extraction operator "[".

```
my.matrix[,3,drop=FALSE]  # Extract a single column, but keep as matrix

##       Gamma
## North     7
## East      9
## South     5
## West      4

dim(my.matrix[,3,drop=FALSE]) # A matrix with just one column.

## [1] 4 1
```

Extracting by name

Extracting rows or columns by number can be very unreliable if you have many columns and the dataset changes from time to time. Your client adds a single new variable and all your code breaks (but you might not even realize it).
Extracting by name can be much more robust.

```
my.matrix[,c("Alpha","Gamma")]

##       Alpha Gamma
## North     5     7
## East      2     9
## South     4     5
## West      3     4

my.matrix["West",c("Alpha","Gamma"),drop=FALSE]

##      Alpha Gamma
## West     3     4
```

Multi-way arrays

Arrays are generalizations of vectors and matrices to more than two dimensions.

```
my.array <- array(c(1:24),dim = c(2,4,3)) # Three, 2 x 4 matrices
print(my.array)

## , , 1
##
##      [,1] [,2] [,3] [,4]
## [1,]    1    3    5    7
## [2,]    2    4    6    8
##
## , , 2
##
##      [,1] [,2] [,3] [,4]
## [1,]    9   11   13   15
## [2,]   10   12   14   16
##
## , , 3
##
##      [,1] [,2] [,3] [,4]
```

Key functions for matrices

Matrix multiply is %*%, transpose is the single letter `t`. For a square matrix, the inverse is `solve`. Arithmetic operators will work elementwise.

```
a <- matrix(c(1,3,5,7),ncol=2); b <- matrix(c(4,2,5,3,1,5),ncol=3)
a %*% b  # Matrix multiply (dimensions must be compatable)

##      [,1] [,2] [,3]
## [1,]   14   20   26
## [2,]   26   36   38

t(b)       # Matrix transpose

##      [,1] [,2]
## [1,]    4    2
## [2,]    5    3
## [3,]    1    5
```

More about matrices

```
solve(a) # matrix inverse

##        [,1]   [,2]
## [1,] -0.875  0.625
## [2,]  0.375 -0.125

#Many functions will work elementwise on a matrix
log(a)   # The log of each element in the matrix

##          [,1]     [,2]
## [1,] 0.000000 1.609438
## [2,] 1.098612 1.945910

a > 4    # comparison operator applied to each element

##       [,1] [,2]
## [1,] FALSE TRUE
## [2,] FALSE TRUE
```

The first of the apply functions

The function `apply` can be used to summarize the rows or columns of an array.

```
print(b)

##      [,1] [,2] [,3]
## [1,]    4    5    1
## [2,]    2    3    5

# Apply the function "sum" to the rows of the matrix
apply(b,   # the matrix
          MARGIN = 1,   # The margin,  1 for rows, 2 for columns
          FUN = sum)    # The chosen function to apply, here "sum"

## [1] 10 10

# Apply the function "mean" to each column.
apply(b,2,mean)

## [1] 3 4 3
```

Lists

Lists are containers that can hold objects of **different** types. It is a good idea to name each object. Extraction is done with one of three operators `[`, `[[`, `$`.

```
# A list with a vector, a matrix and a logical vector
my.list <- list(x = seq(10), y = matrix(rnorm(16),ncol=4),
                z = rep(c(TRUE,FALSE),c(3)))
# Pull out the second element
my.list[2]    # But it is still a list!

## $y
##           [,1]          [,2]        [,3]       [,4]
## [1,] 1.8675474 -0.1627264369  0.01155596  0.4591213
## [2,] 1.3335873 -0.2972345230 -0.10367355  0.8822012
## [3,] 0.1958973  0.0009649303  0.42132556  1.8660353
## [4,] 0.3700970 -1.1436643776 -0.04462218 -0.3466730

class(my.list[2])

## [1] "list"
```

Lists

To get directly at the list element itself use `[[`

```
my.list[[2]] # Is the actual matrix itself, note its class below

##           [,1]          [,2]        [,3]       [,4]
## [1,] 1.8675474 -0.1627264369  0.01155596  0.4591213
## [2,] 1.3335873 -0.2972345230 -0.10367355  0.8822012
## [3,] 0.1958973  0.0009649303  0.42132556  1.8660353
## [4,] 0.3700970 -1.1436643776 -0.04462218 -0.3466730

class(my.list[[2]])

## [1] "matrix"
```

Lists

Accessing list elements by name:

```
my.list["z"]   # Accessing by name (note the quotes)

## $z
## [1]  TRUE FALSE  TRUE FALSE  TRUE FALSE

my.list[["z"]]   # Accessing by name (note the quotes)

## [1]  TRUE FALSE  TRUE FALSE  TRUE FALSE

my.list$z    # Or, another way of accessing a list element by name

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

The lapply function

The `lapply` function will apply a function to the elements of a list, and return another list with the output of the function.

```
lapply(my.list,   # The list
       FUN = sum) # The function I am applying to each element (sum)

## $x
## [1] 55
##
## $y
## [1] 5.309739
##
## $z
## [1] 3
```

What's the difference between [, [[and $?

- [let's you extract more than 1 element at a time and returns a list.
- `[[` and `$` will only let you extract a single element.

```
my.list[c(1,3)]   # OK

## $x
##  [1]  1  2  3  4  5  6  7  8  9 10
##
## $z
## [1]  TRUE FALSE  TRUE FALSE  TRUE FALSE

my.list[[c(1,3)]]    # Not what you thought!

## [1] 3

# In fact, this is the third entry in the first element of the list.
# So called "recursive extraction".
```

List element replacement

Similar to vectors and matrices:

```
my.list[1] <- "Hello"     # Replace the first element
my.list["d"] <- "Goodbye" # Add an element called "d"
my.list$e <- "That's it!" # Add another called "e"

names(my.list) # Returns all the names in the list

## [1] "x" "y" "z" "d" "e"

names(my.list)[3] <- "Logical array" # Overwrite a name
names(my.list)

## [1] "x"             "y"             "Logical array"
## [4] "d"             "e"
```

Uses of lists

Many R functions return their output as a list. For example the regression function `lm`.

```
x <- rnorm(100); y <- rnorm(100)  # A predictor and outcome variable
lm.out <- lm(y ~ x) # Our first regression model in R!
#lm.out contains the output from the regression model
class(lm.out) # What class is the output?

## [1] "lm"

attributes(lm.out) # This function returns the attributes of an object

## $names
##  [1] "coefficients"  "residuals"     "effects"
##  [4] "rank"          "fitted.values" "assign"
##  [7] "qr"            "df.residual"   "xlevels"
## [10] "call"          "terms"         "model"
##
## $class
## [1] "lm"
```

Uses of lists

Remembering that this is just a list, shows how we can access the various elements

```
lm.out["coefficients"]  # A list with a single element, the coefficients

## $coefficients
## (Intercept)           x
## -0.02804920 -0.06520162

lm.out[["coefficients"]] # The actual coefficients themselves

## (Intercept)           x
## -0.02804920 -0.06520162

lm.out$coefficients     # The common way to get at the coefficients

## (Intercept)           x
## -0.02804920 -0.06520162
```

Uses of lists

- One of the ways I like to use lists is to store meta information about other objects.
- In a survey we may have multiple questions and want to give different questions different attributes.
- I store the meta data about the questions in a big list I call the "codebook", which is in fact a list of lists.
- Ultimately we cycle through the list, mixing the meta data with the raw data to create the report.

Uses of lists

```r
codebook <- list() # Start with an empty list

#Now start adding question meta info to it.

codebook[["q1"]] <- list(LENGTH=3,
                         LONGNAME="Purchase activity",
                         SHORTNAME="Purchase")
codebook[["q2"]]  <- list(LENGTH=2,
                         LONGNAME="Make of purchased vehicle",
                         SHORTNAME="Make")
codebook[["q3"]]  <- list(LENGTH=3,
                         LONGNAME="I attend this Auto Show",
                         SHORTNAME="Attendance")
codebook[["q4"]]  <- list(LENGTH=3,
                         LONGNAME="I decided to attend the Auto Show",
                         SHORTNAME="Attend decision")
```

Uses of lists

Each question is given three pieces of information.

- `LENGTH` is an integer, 2 or 3 which tells me how to summarize the question, (graphically and numerically).
- `LONGNAME` will appear in the heading of the question summary.
- `SHORTNAME` will appear in the table of contents.

The names of the elements in the list are of my choosing. Unusually, as you can see above, you don't in fact have to quote the names when you **define** the list (LENGTH,LONGNAME,SHORTNAME) are not quoted here. It doesn't make a difference in this particular context.

Uses of lists

```
codebook[["q4"]] # All the information about a question

## $LENGTH
## [1] 3
##
## $LONGNAME
## [1] "I decided to attend the Auto Show"
##
## $SHORTNAME
## [1] "Attend decision"

codebook[["q4"]][[2]] # The actual question text itself

## [1] "I decided to attend the Auto Show"

# Note you can chain the [[ operator to drill further into the structure
```

Module summary

Topics covered today include:

- Vectors
- Matrices and arrays
- Extraction and replacement
- The apply and lapply functions

Next time

- The `dataframe` structure
- Reading data into R
- Merging data

R gotchas

- If you mix different `types` of elements in a vector, they will get coerced to the same type, typically all character.
- Not realizing that if the number of elements in the logical extraction vector does not equal the number of elements in the vector you are extracting from, then R will **recycle** the logical vector. This can lead to you getting the wrong elements by accident
- When subsetting a single column or row from a matrix, the results will be a vector, not a matrix, unless you use the drop=FALSE argument to the extraction operator
- Using the single letters `c` or `t` as variable names. These are both functions in R, concatenation and transpose respectively

Today's function list

Do you know what each of these functions does?

```
apply        t
attributes   [
c            [[
class        $
colnames     %*%
dim
lapply
length
letters
lm
matrix
names
rep
rownames
seq
solve
```