Introduction to R programming

author: Rebecca C. Steorts date: 17 January 2017

Agenda

- Using R, RStudio, Markdown
- Functional programming
- Vectors
- Example on housing prices
- Matrices
- Lists
- Dataframes

Reproducible Research

Reproducible research is the idea that data analyses, and more generally, scientific claims, are published with their data and software code so that others may verify the findings and build upon them.

-Johns Hopkins, Coursera

The R Console

Basic interaction with R is by typing in the **console**, a.k.a. **terminal** or **command-line**

You type in commands, R gives back answers (or errors)

Menus and other graphical interfaces are extras built on top of the console

RStudio

- RStudio is very easy and simple to use. It can be downloaded from R Studio Download (https://www.rstudio.com).
- RStudio is not R.
- RStudio mediates your interaction with R.

What is Markdown?

- Markdown is a lightweight markup language for creating HTML, PDF, or other documents.
- Markup languages are designed to produce documents from human readable text.
- This promotes research/materials that are reproducible.

- Also, RStudio integrates with LaTeX.

Why Markdown?

- It's easy to learn.
- It really pushes at reproducible code and documentation.
- Once this basics are down, you can do things that are more fancy.

Getting started with RStudio + Markdown

- A cheatsheet is given for simple markdown commands R Markdown Cheat Sheet (https://www.rstudio.com/wp-content/uploads/2015/02/rmarkdown-cheatsheet.pdf)
- Tysetting equations can be slightly different than LaTeX. There are some resources here! LaTeX Typesetting (http://www.statpower.net/Content/310/R%20Stuff/SampleMarkdown.html)

Simple Illustration in RStudio

```
1+6
```

```
## [1] 7
```

```
x <- 4
(x + 2)
```

```
## [1] 6
```

```
set.seed(738)
```

Data in R

Most variables are created with the **assignment operator**, `<-` or `=`

```
average.rent.dollar <- 800
average.rent.dollar
```

```
## [1] 800
```

```
dollar.to.euro = 0.93
average.rent.dollar*dollar.to.euro
```

```
## [1] 744
```

=== The assignment operator also changes values:

```
average.rent.euro <- average.rent.dollar*dollar.to.euro
average.rent.euro
```

```
## [1] 744
```

```
average.rent.euro <- 744
average.rent.euro
```

```
## [1] 744
```

The workspace

What names have you defined values for?

```
ls()
```

```
## [1] "average.rent.dollar" "average.rent.euro"   "dollar.to.euro"
## [4] "x"
```

Getting rid of variables:

```
rm("average.rent.euro")
ls()
```

```
## [1] "average.rent.dollar" "dollar.to.euro"      "x"
```

Review of vectors

Group related data values into one object, a **data structure**

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

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

`c()` function returns a vector containing all its arguments in order

`x[1]` is the first element, `x[3]` is the 3rd element `x[-3]` is a vector containing all but the 3rd element

Question: What does x[-c(2:3)] return?

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

```
weekly.hours <- vector(length=5)
weekly.hours[5] <- 8
weekly.hours
```

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

=== Operators apply to vectors “pairwise” or “elementwise”:

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

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

```
x*y
```

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

=== Can also do pairwise comparisons:

```
x > 9
```

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

Note: returns Boolean vector

Boolean operators work elementwise:

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

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

Functions on vectors

Many 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 `7 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:

```
x
```

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

```
y
```

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

```
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
```

Example: Price of houses in PA

Census data for California and Pennsylvania on housing prices, by Census “tract”

```
calif_penn <-read.csv("http://www2.stat.duke.edu/~rcs46/modern_bayes17/data/calif_pen
n_2011.csv")
penn <- calif_penn[calif_penn[,"STATEFP"]==42,]
coefficients(lm(Median_house_value ~ Median_household_income, data=penn))
```

```
##             (Intercept) Median_household_income
##           -26206.564325                3.651256
```

Fit a simple linear model, predicting median house price from median household income

=== Census tracts 24–425 are Allegheny county

Tract 24 has a median income of $14,719; actual median house value is $34,100 — is that above or below the observed median?

```
34100 < -26206.564 + 3.651*14719
```

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

Tract 25 has income $48,102 and house price $155,900

```
155900 < -26206.564 + 3.651*48102
```

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

What about tract 26?

===

We *could* just keep plugging in numbers like this, but that’s - boring and repetitive - error-prone - confusing (what *are* these numbers?)

Using variables and names

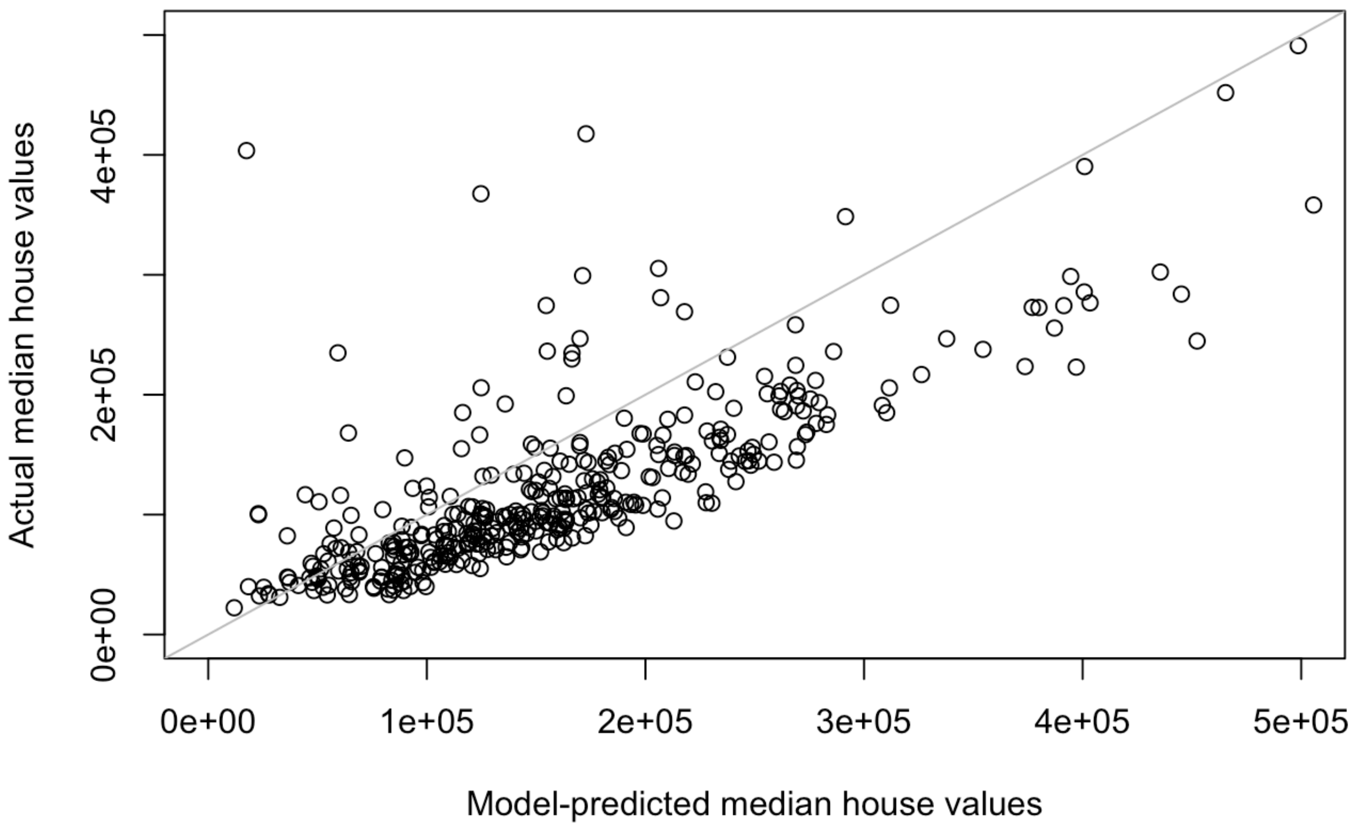
```
penn.coefs <- coefficients(lm(Median_house_value ~ Median_household_income, data=penn
))
penn.coefs
```

```
##             (Intercept) Median_household_income
##           -26206.564325                3.651256
```

```
allegheny.rows <- 24:425
allegheny.medinc <- penn[allegheny.rows,"Median_household_income"]
allegheny.values <- penn[allegheny.rows,"Median_house_value"]
allegheny.fitted <- penn.coefs["(Intercept)"]+penn.coefs["Median_household_income"]*a
llegheny.medinc
```

===

```
plot(x=allegheny.fitted, y=allegheny.values,
     xlab="Model-predicted median house values",
     ylab="Actual median house values",
     xlim=c(0,5e5),ylim=c(0,5e5))
abline(a=0,b=1,col="grey")
```



Simple example: resource allocation (“mathematical programming”)

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 proceed as usual (e.g., `factory/5`)

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
```

Exercise: What 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
```

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 character vectors

We use the same function to get and to set their respective values

Names are useful since they help us keep track of what we are working with

===

```
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("cars","trucks")
factory %*% output
```

```
##       [,1]
## labor 1400
## steel   50
```

```
factory %*% output[colnames(factory)]
```

```
##       [,1]
## labor 1400
## steel   50
```

```
all(factory %*% output[colnames(factory)] <= available[rownames(factory)])
```

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

Summaries

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 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?

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 (cont'd.)

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"]] <- "2011-08-30"
```

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 `penn` from last time was really a dataframe

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
```

```
a.data.frame[1,]
```

```
##   v1 v2 logicals
## 1 35 10     TRUE
```

```
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
```

Structures of Structures

So far, every list element has been a single data value

List elements can be other data structures, e.g., vectors and matrices:

```
plan <- list(factory=factory, available=available, output=output)
plan$output
```

```
##   cars trucks
##     20     10
```

Internally, a dataframe is basically a list of vectors

Structures of Structures (cont’d.)

List elements can even be other lists which may contain other data structures including other lists which may contain other data structures…

This **recursion** lets us build arbitrarily complicated data structures from the basic ones

Most complicated objects are (usually) lists of data structures

Take-Aways

- Write programs by composing functions to manipulate data
- The basic data types let us represent Booleans, numbers, and characters
- Data structures let us group related values together
- Vectors let us group values of the same type

- Use variable assignment and name components of structures to make data more meaningful
- 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