MFE Programming Workshop Class

Robert J. Richmond

November 2, 2015

UCLA Anderson

Goals

- Learn to program in R and in Matlab
- What does programming mean?
    - Language syntax
    - Debugging
    - Finding solutions
    - Translating math to code
- This is just the beginning, you'll develop these skills throughout the program

R vs Matlab

- Both are useful and you will use both in the MFE program
- My view:
  - R is good for data munging, statistics, regressions, etc.
  - Matlab is good for simulations, numerical solvers, etc.
- This workshop will demonstrate these differences

Structure

- I will talk for 30-60 minutes at the beginning of each class
- For the remainder of the time you will break into groups and work on programming tasks
- Tasks are designed to introduce you to the building blocks that will be used for course assignments throughout the MFE program
- This course is a programming course with emphasis on methods for finance:
    - You *will* see finance terms and math
    - You *may* not understand all of the finance, but you will learn it throughout the program
- The key skills will be translating mathematical algorithms into code and developing the ability to find helpful resources

Questions

Any questions before we start?

R Environment

- R Studio is a fantastic environment to interact with R
- I am going to assume that you have a working installation of R Studio and that you have a basic understanding of how it works
- My focus is going to be on R programming

R References

- See list on Rossi's page
- The Art of R Programming by Norman Matloff
  - This is a fantastic book and it is the primary source for these lectures
- Built in documentation!
  - `?funcname`
  - Like in Matlab, this is the most useful reference. Learn to read and understand the built in documentation
- R Cookbook is useful for finding specific code snippets

R as a language

- R is object oriented
  - Everything is an object and functions operate differently when passed different types of objects
- R is functional
  - You write fewer loops
  - You write cleaner code

Vectors and Assignment

- Assigning values to variables can be done with `<-`
  - `=` works, but there are reasons to prefer `<-`
- Create vectors of numbers using the function `c()`

Example

```
myvector <- c(1,2,3,4)
samevector <- 1:4
vectorsubset <- myvector[1:3]
vectorsubset
```

```
## [1] 1 2 3
```

c() function

- c, which stands for concatenate is a very flexible function

Example

```
myvec1 <- 2:4
myvec2 <- c(1,3,5,myvec1)
myvec2
```

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

Accesing elements of vectors

- Elements can be accessed using `[ ]`

Example

```
myvec <- c(2,4,6,8)
myvec[4]

## [1] 8

myvec[c(1,3)]

## [1] 2 6
```

Length

- `length()` returns the vector length

Example

```
myvec <- 1:23
length(myvec)


## [1] 23
```

Recycling

- Vectors are recycled when an operation acts elementwise
- Be careful with and aware of this behavior!
- In some cases it is useful, others confusing

Example

```
vec1 <- 1
vec2 <- 1:4
vec3 <- 1:2
vec2+vec1
```

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

```
vec2+vec3
```

seq and rep

- Useful functions for generating vectors
- See `?seq` and `?rep` for details

Example

```
myvec1 <- seq(1,10,2)
myvec1
```

```
## [1] 1 3 5 7 9
```

```
myvec2 <- rep(c(1,2),3)
myvec2
```

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

NULL and NA

- `NULL` is the non-existent value in R
- `NA` is the missing place hold

Example

```
myvec1 <- 5:8
myvec1[2] <- NA
myvec1

## [1]  5 NA  7  8

myvec2 <- NULL
length(myvec2)

## [1] 0
```

Filtering (1)

- Select subsets using vectors of logicals

Example

```
vec1 <- 1:5
vec2 <- c(TRUE,FALSE,TRUE,FALSE,TRUE)
vec1[vec2]
```

```
## [1] 1 3 5
```

Filtering (2)

- Select subsets using vectors of logicals

Example

```
vec1 <- 1:6
vec2 <- vec1 > 3
vec2

## [1] FALSE FALSE FALSE  TRUE  TRUE  TRUE

vec1[vec2]

## [1] 4 5 6

vec1[vec1>3]
```

Assigning to filter

- You can assign to the subsets

Example

```
vec1 <- 1:6
vec1[vec1<2] <- NA
vec1
```

```
## [1] NA  2  3  4  5  6
```

Functions on vectors

- Functions typically operate on vectors

Example

```
x <- 1:1000
mean(x)

## [1] 500.5
```

names

- You can give names to elements of vectors

Example

```
myvec <- 1:3
names(myvec) <- c("A","B","C")
myvec["B"]


## B
## 2
```

Intro

- Matrices are vectors with a number of rows and number of columns attribute

Example

```
myvec <- 1:10
mymat <- matrix(myvec, nrow=2, ncol=5)
mymat

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

mymat[1,3]

## [1] 5
```

Matrix operations

- Many matrix operations are surrounded by % signs

Example

```
mymat1 <- matrix(1:4, nrow=2)
mymat2 <- matrix(5:8, nrow=2)
mymat1 %*% mymat2
```

```
##      [,1] [,2]
## [1,]   23   31
## [2,]   34   46
```

```
mymat1 + mymat2
```

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

apply

- `apply` allows you to apply a function across a dimension of a matrix
- The third argument is a function!

Example

```
mymat1 <- matrix(1:10, nrow=2)
# mean across rows
apply(mymat,1,mean)

## [1] 5 6
```

cbind and rbind

- Column bind and Row bind

Example

```
mymat1 <- matrix(1:4,nrow=2)
mymat2 <- matrix(6:9,nrow=2)
mymat3 <- matrix(10:11,ncol=2)
cbind(mymat1,mymat2)
```

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

```
rbind(mymat1,mymat3)
```

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

Intro

- Lists are where you really start to see the advantages of R as a statistics and data manipulation language

Example

```
element1 <- 1:5
element2 <- matrix(1:6,nrow=3)
mylist <- list(el1=element1,el2=element2)
mylist["el1"]]

## [1] 1 2 3 4 5

mylist["el2"]]

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

Subsetting lists

- Subsets of lists are with single `[ ]`

Example

```
mylist <- list(A=1,B=2,C=3,D=4)
# this returns a list because of the single []
mylist[c(1,3)]
```

```
## $A
## [1] 1
##
## $C
## [1] 3
```

lapply

- `lapply` implicitly loops over each list element and applies a function

Example

```
mylist <- list(A=1:10,B=2:17,C=745:791)
lapply(mylist,mean)
```

```
## $A
## [1] 5.5
##
## $B
## [1] 9.5
##
## $C
## [1] 768
```

lapply example

```
g <- c("M","F","F","I","M","M","F")
lapply(c("M","F","I"),function(gender) which(g==gen


## [[1]]
## [1] 1 5 6
##
## [[2]]
## [1] 2 3 7
##
## [[3]]
## [1] 4
```

Intro

- In my mind, `data.frame` is the core data type in R
- The nicest part is that they can hold different types

Example

```
dat1 <- 1:4
dat2 <- rep(c("A","B"),each=2)
myframe <- data.frame(col1=dat1,col2=dat2)
myframe
```

```
##   col1 col2
## 1    1    A
## 2    2    A
## 3    3    B
## 4    4    B
```

Subsets

- Subsetting is just like a matrix

Example

```
myframe[1,2]

## [1] A
## Levels: A B

myframe[1,]

##   col1 col2
## 1    1    A

myframe[,2]
```

Reading in data

- Reading in data typically gives you a `data.frame`
- `read.table` is the basic function to read in tabular data
- `read.csv` is a special cast of `read.table`
- As usual see `?read.table`

Adding columns

Example

```
dat1 <- 1:4
dat2 <- rep(c("A","B"),each=2)
myframe <- data.frame(col1=dat1,col2=dat2)
myframe$col3 <- 5:8
myframe

##   col1 col2 col3
## 1    1    A    5
## 2    2    A    6
## 3    3    B    7
## 4    4    B    8
```

names

- column names in data.frames are specificed by `names()`
- this is because data.frames are actually lists with special attributes
- that means that the usual list functions work on data.frames
    - lapply, etc

Long example

Example

```
all2006 <- read.csv("2006.csv",header=TRUE,as.is=TR
all2006 <- all2006[all2006$Wage_Per=="Year",] # exc
all2006 <- all2006[all2006$Wage_Offered_From > 2000
all2006$rat <- all2006$Wage_Offered_From / all2006$
se2006 <- all2006[grep("Software Engineer",all2006)
```

For loops (1)

Example

```
x <- c(1:5)
y <- NULL
for(i in 1:length(x)) {
   y[i] <- x[i] + 2
}
y


## [1] 3 4 5 6 7
```

For loops (2)

- or another nice way to make a for loop

Example

```
x <- c(1:3)
for(element in x) {
   print(element + 2)
}


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

While loops

Example

```
x <- c(1:5)
y <- NULL
i <- 1
while(i<=length(x)) {
   y[i] <- x[i] + 2
   i <- i+1
}
y


## [1] 3 4 5 6 7
```

Conditional Statements

Example

```
x <- -10
myabs <- x
if(x<0) {
    myabs <- -x
}
myabs

## [1] 10
```

Function definitions

- Note that the last value evaluated is what is returned by the function

Example

```
myfunc <- function(x) x^2
myfunc(10)

## [1] 100
```

Scope rules

- Variables defined inside a function are local to that function

Example

```
myfunc <- function(x) {
    N <- 10
    N*x^2
}
myfunc(10)

## [1] 1000

# You can't access N out here
```