MFE R Programming Workshop

Week 3

Brett Dunn and Mahyar Kargar

Fall 2017

Introduction

Questions

Any questions before we start?

Overview of Week 3

- We will cover four classes to store array-like data:
  - Matrices
  - Data Frames
  - Tibbles
  - Intro to Data Tables

Changing the Working Directory

- Use the `setwd()` R function
- Use the **Tools** | **Global Options** | **General** menu
- From within the Files pane, use the **More** | **Set As Working Directory** menu. (Navigation within the Files pane alone will not change the working directory.)

Matricies vs Data Frames

- Matrix is a data type in R with the dimension attribute - the rows and the columns.
  - It has the elements of *same* class type.
  - We can have character, integer or complex elements in the matrices and so on.
  - We *cannot* have elements of mixed modes/class types such as both integer and character elements in the same matrix.
- A `data.frame` is a list of vectors of equal length, and the vectors can be of different types.
  - e.g. one character column, one numeric column.
- Tibbles and data tables inherit the functionality of data frames and improve on them in various ways.

Matrices

Creating Matrices

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

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

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

```
dim(mymat)  # returns the dimension
```

```
## [1] 2 5
```

Accessing Elements of Matrices

- Like vectors, elements can be accessed using []

```
mymat <- matrix(1:15, nrow=3, ncol=5)
mymat[1, 2]  # row 1, column 2
```

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

```
mymat[2:3, c(1, 4, 5)]
```

```
##      [,1] [,2] [,3]
## [1,]    2   11   14
## [2,]    3   12   15
```

Filtering Matrices

- Filtering can be done on a single column or a single row, otherwise the filter returns a vector.

```
myvec <- c(1, 1, 3, 1, 5, 1, 7, 1, 9, 1)
mymat <- matrix(myvec, nrow=2, ncol=5)
mymat
```

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

```
mymat[, mymat[1, ] > 4]
```

```
##      [,1] [,2] [,3]
## [1,]    5    7    9
## [2,]    1    1    1
```

Vectorization

- Most R functions work on matrices as well.

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

```
##          [,1]     [,2]     [,3]     [,4]      [,5]
## [1,] 2.718282 20.08554 148.4132 1096.633  8103.084
## [2,] 7.389056 54.59815 403.4288 2980.958 22026.466
```

```
sd(mymat)  # standard deviation
```

```
## [1] 3.02765
```

Applying Functions to Rows and Columns

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

```r
mymat <- matrix(1:10, nrow=2)
# mean across rows (can also use rowMeans() function)
apply(mymat, 1, mean)  # apply mean along rows
```

```
## [1] 5 6
```

```r
apply(mymat, 2, max)  # apply max along columns
```

```
## [1]  2  4  6  8 10
```

Combining Matricies with cbind and rbind

- Column bind and row bind.

```
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]
## [1,]    1    3
## [2,]    2    4
## [3,]   10   11
```

Matrix operations

- Many matrix operations are surrounded by % signs.

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

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

```
mymat1 + mymat2
```

```
##      [,1] [,2]
## [1,]    6   10
## [2,]    8   12
```

Matrix Algebra

- http://www.statmethods.net/advstats/matrix.html

Data Frames

data.frames

- The data.frame is one of the most useful features in R.
- A data.frame is like a matrix with a two-dimensional rows-and-columns structure.
- However, unlike a matrix, in a data.frame each column can have a *different* data type.
    - For example, one column might be numbers and another characters.
- Technically, a data.frame is a list, with the components of the list being equal-length vectors.
- Each column must be the same length (unlike a list).

Creating data.frames

- Unless you are working with categorical data, you probably want to set stringsAsFactors=FALSE.

```r
courses <- c("Stochastic Calculus", "Fixed Income")
examGrades <- c(92, 98)
gradeBook <- data.frame(courses, examGrades,
                        stringsAsFactors = FALSE)
gradeBook
```

```
##               courses examGrades
## 1 Stochastic Calculus         92
## 2        Fixed Income         98
```

Column Names

- Column names in data.frames are specified 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.

Accessing Elements of `data.frames`

Accessing Elements of `data.frames`

- We can access a `data.frame` component just like a `list`.

```
gradeBook[[1]]  # fist way
```

```
## [1] "Stochastic Calculus" "Fixed Income"
```

```
gradeBook[["courses"]]  # second way
```

```
## [1] "Stochastic Calculus" "Fixed Income"
```

```
gradeBook$courses  # third way
```

```
## [1] "Stochastic Calculus" "Fixed Income"
```

Accessing Elements of `data.frames` (2)

- Note that `[[` simplifies the result, returning a vector:

```
str(gradeBook[[1]])
```

```
##  chr [1:2] "Stochastic Calculus" "Fixed Income"
```

- But `[` is preserving subsetting and (usually) returns a ‘data.frame:

```
str(gradeBook[1])
```

```
## 'data.frame':    2 obs. of  1 variable:
##  $ courses: chr  "Stochastic Calculus" "Fixed Income"
```

Accessing Elements of data.frames like a matrix

- We can access data.frame elements like a matrix.

```
gradeBook[1,2]
```

```
## [1] 92
```

```
gradeBook[1,] # the first row
```

```
##                courses examGrades
## 1 Stochastic Calculus         92
```

```
gradeBook[,2] # the second column
```

```
## [1] 92 98
```

Accessing Elements of data.frames like a matrix(2)

- To preserve the data.frame class, set drop = FALSE

```
gradeBook[,2] # returns a vector
```

```
## [1] 92 98
```

```
gradeBook[,2,drop=FALSE] # returns a data.frame
```

```
##   examGrades
## 1         92
## 2         98
```

Subsetting `data.frames`

Filtering with subset()

```
set.seed(1234)
x.df <- data.frame(V1 = rnorm(4),V2 = runif(4),
                   V3 = rchisq(4,df = 2), V4 = 1:4)
x.df
```

```
##           V1        V2        V3 V4
## 1 -1.2070657 0.6660838 0.3523580  1
## 2  0.2774292 0.5142511 2.8742845  2
## 3  1.0844412 0.6935913 0.3134394  3
## 4 -2.3456977 0.5449748 0.4390040  4
```

```
x.sub <- subset(x.df, V4 > 2)
x.sub
```

```
##          V1        V2        V3 V4
## 3  1.084441 0.6935913 0.3134394  3
## 4 -2.345698 0.5449748 0.4390040  4
```

Subsetting rows using conditional statements

- The data frame `x.sub1` contains only the observations for which the values of the variable `V4` is greater than 2 and the variable `V1` is greater than 0.6.

```
x.sub1 <- subset(x.df, V4 > 2 & V1 > 0.6)
x.sub1
```

```
##         V1        V2        V3 V4
## 3 1.084441 0.6935913 0.3134394  3
```

Subsetting both rows and columns

- The data frame x.sub2 contains only the variables V2 and V3 and then only the observations of these two variables where the values of variable V4 are greater than 2 and the values of variable V1 are greater than 0.6.

```
x.sub2 <- subset(x.df, V4 > 2 & V1 > 0.6,
                 select = c(V2, V3))
x.sub2
```

```
##          V2        V3
## 3 0.6935913 0.3134394
```

Subsetting rows using indices

- The `x.sub3` data frame contains only the observations for which the values of variable `V4` are equal to 2.

```
x.sub3 <- x.df[x.df$V4 == 2, ]
x.sub3
```

```
##          V1        V2       V3 V4
## 2 0.2774292 0.5142511 2.874284  2
```

Subsetting rows using %in%

- The x.sub4 data frame contains only the observations for which the values of variable V4 are equal to either 1 or 4.

```
x.sub4 <- x.df[x.df$V4 %in% c(1, 4), ]
x.sub4
```

```
##          V1        V2       V3 V4
## 1 -1.207066 0.6660838 0.352358  1
## 4 -2.345698 0.5449748 0.439004  4
```

Subsetting columns using indices

- The x.sub5 data frame contains all the rows on x.df, removing the first and third columns

```
x.sub5 <- x.df[, -c(1,3)]
x.sub5
```

```
##          V2 V4
## 1 0.6660838  1
## 2 0.5142511  2
## 3 0.6935913  3
## 4 0.5449748  4
```

Complete.cases

- `complete.cases()` gets rid of any *rows* with at least one `NA` value.

```r
# Let's makes the second col in row 1 an NA
x.df[1,2] <- NA
x.df[complete.cases(x.df), ] # removes row 1
```

```
##           V1        V2        V3 V4
## 2  0.2774292 0.5142511 2.8742845  2
## 3  1.0844412 0.6935913 0.3134394  3
## 4 -2.3456977 0.5449748 0.4390040  4
```

Merging data.frames

- Two data.frames can be combined using the merge function.

```
courses <- c("Stochastic Calculus", "Fixed Income")
midtermGrades <- c(89, 91)
gradeBook2 <- data.frame(courses, midtermGrades,
                         stringsAsFactors = FALSE)
merge(gradeBook, gradeBook2)
```

```
##                 courses examGrades midtermGrades
## 1          Fixed Income         98            91
## 2 Stochastic Calculus         92            89
```

Adding Columns to data.frames

```r
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
```

Reading in Data from a CSV File

- 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 case of `read.table`.
- As usual see `?read.table`.
- Often you want to set `stringsAsFactors = FALSE`.
- `write.csv` writes data to a .csv file.

```r
optdata <- read.csv(file="../week2/lab/optionsdata.csv",
                header = T, stringsAsFactors = FALSE)
head(optdata, 3)
```

```
##     S0 sigma   r T   K
## 1 100   0.3 0.0 1 100
## 2 101   0.3 0.0 1 100
## 3 101   0.1 0.1 1 105
```

tibbles

What is a `tibble`?

- Tibbles are a “modern take” on R’s traditional `data.frame`.
- They keep the features that have stood the test of time, and drop the features that used to be convenient but are now frustrating (i.e. converting character vectors to factors).

Creating a `tibble`

- `tibble()` can be used to create a data frame.
- It never changes an input's type (i.e., no more `stringsAsFactors = FALSE`!).

```
tibble(x = letters)
```

```
## # A tibble: 26 × 1
##         x
##     <chr>
## 1       a
## 2       b
## 3       c
## 4       d
## 5       e
## 6       f
## 7       g
## 8       h
## 9       i
```

Creating a tibble of lists

This makes it easier to use with list-columns:

```
tibble(x = 1:3, y = list(1:5, 1:10, 1:20))
```

```
## # A tibble: 3 × 2
##       x          y
##   <int>     <list>
## 1     1  <int [5]>
## 2     2 <int [10]>
## 3     3 <int [20]>
```

Lazy and Sequential Evaluation

It evaluates its arguments lazily and sequentially:

```
tibble(x = 1:5, y = 1, z = x ^ 2 + y)
```

```
## # A tibble: 5 × 3
##       x     y     z
##   <int> <dbl> <dbl>
## 1     1     1     2
## 2     2     1     5
## 3     3     1    10
## 4     4     1    17
## 5     5     1    26
```

Column Names

- Tibbles never adjust the names of variables:

```
names(data.frame(`crazy name` = 1))
```

```
## [1] "crazy.name"
```

```
names(tibble(`crazy name` = 1))
```

```
## [1] "crazy name"
```

Other Features

- Tibbles don't use `row.names()`. It never stores a variable as special attribute.
- Tibbles only recycle vectors of length 1. This is because recycling vectors of greater lengths is a frequent source of bugs.
- Tibble provides `as_tibble()` to coerce objects into tibbles.

Tibbles vs Data Frames: Printing

- When you print a tibble, it only shows the first ten rows and all the columns that fit on one screen. It also prints an abbreviated description of the column type.
- You can control the default appearance with options.

```r
options(tibble.print_max = 3, tibble.print_min = 2)
tibble(x = 1:1000)
```

```
## # A tibble: 1,000 × 1
##        x
##    <int>
## 1      1
## 2      2
## # ... with 998 more rows
```

Tibbles vs Data Frames: Subsetting

- If you want to pull out a single variable, you could use $ and [[or %>% ([[can extract by name or position; $ only extracts by name)

```
df <- tibble( x = 1:5, y = rnorm(5))
# Extract by name (df[["x"]] works the same way)
df$x
```

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

```
# Extract by position
df[[1]]
```

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

```
df %>% .$x # df %>% .[["x"]] works the same way
```

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

Tibbles vs Data Frames: Subsetting with [[and $

- Recall that for data frames:
  - `[[` extracts a single column as a vector.
  - `$` works similarly to `[[`, but does *partial matching* on the column name.

```r
df <- data.frame(colName = 1:5, m = 2:6); df$c
```

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

- Tibbles never do partial matching, and will throw a warning and return `NULL` if the column does not exist.

```r
tbl <- as_tibble(df); tbl$c
```

```
## Warning: Unknown or uninitialised column: 'c'.
```

```
## NULL
```

Tibbles vs Data Frames: Recycling

- When constructing a `tibble`, only values of length 1 are recycled.
- The first column with length different to one determines the number of rows in the tibble, conflicts lead to an error.
  - `tibble(a = 1:3, c = 1:2)` gives "`Error: Column c must be length 1 or 3, not 2`".

Very quick intro to `data.tables`

What is a data.table?

- Think of data.table as an advanced version of data.frame.
  - Every column is the same length, but may have a different type
- It inherits from data.frame and works even when data.frame syntax is applied on data.table
- data.table is *very fast*.
- It is one of the most useful packages in R.
- The syntax of data.table is concise.
  - Lowers programmer time…
  - …but it can be hard to understand
  - Make sure you comment your code!

```r
library(data.table)
```

An Example

- Syntax is `DT[i, j, by]`:
- "Take DT, subset rows using `i`, then calculate `j` grouped by `by`."

```
data("mtcars")
mtcarsDT <- data.table(mtcars)
mtcarsDT[ mpg > 20,
          .(AvgHP = mean(hp), "MinWT(kg)" = min(wt*453.6)),
          by = .(cyl, under5gears = gear < 5)]
```

```
##     cyl under5gears     AvgHP MinWT(kg)
## 1:    6        TRUE 110.00000 1188.4320
## 2:    4        TRUE  78.33333  732.5640
## 3:    4       FALSE 102.00000  686.2968
```

Why learn `data.table`?

- For data that fits in memory, `data.table` is much faster than `data.frames` and `tibbles`.
- `data.table` also saves memory because it avoids copying large objects.
- Part of the speed advantage comes from the fact that `data.table` provides a set of tools to update by reference.
  - In base `R`, if a function modifies a single element of a large `data.frame`, a copy of the entire `data.frame` is made.
- `data.table` provides a powerful set of commands to access data.
- We will cover `data.table` in much more detail later in the course.