

MFE R Programming Workshop

Week 3

Brett R. Dunn

Fall 2016

Introduction

Questions

Any questions before we start?

Overview of Week 2

- ▶ Some random R tricks.
- ▶ Importing Data from Databases
- ▶ Importing Data from the Web.
- ▶ Importing Data from WRDS
- ▶ `data.table`

Random R Tricks

Send an Email from R

```
library(gmailr)
hiFromR <- mime(
  To = "someone@anderson.ucla.edu",
  From = "me@gmail.com",
  Subject = "hello from R",
  Body = "I am writting you from R."
) %>%
attach_file("./file.R")
send_message(hiFromR)
```

Working with Files in R

```
dir.exists("week3") # does the directory exist?  
dir.create("week4") # create a directory  
# download files  
download.file(url, destinationfile, method='curl')  
untar() # untar a file  
tar() # tar a file  
# display file information  
f<-(dir("/path/to/something", full.names = TRUE) %>% file.info())  
# delete a file (be careful!!)  
unlink("/path/to/something")
```

Importing Data from Databases

Databases

- ▶ A database stores data in table format.
- ▶ There are several popular database management systems (DBMS):
 - ▶ MySQL, PostgreSQL, SQLite, Oracle, Microsoft SQL Server, etc.
- ▶ Structured Query Language (SQL) is used to maintain and query the database.

Databases in R

- ▶ There are different R packages for each DMS:
 - ▶ MySQL = RMySQL
 - ▶ PostgreSQL = RPostgreSQL
 - ▶ SQLite = RSQLite
 - ▶ Oracle = ROracle
 - ▶ SQL Server = RSQLServer
- ▶ The functions we use to interact with the database are specified in the R package DBI.

A MySQL Example: Establish a Connection

- First, we need to establish a connection to the database.

```
library(DBI)
con <- dbConnect(RMySQL::MySQL(), # the MySQL driver
                 dbname = "dbname", # database name
                 host = "IP_or_WebAddress",
                 port = 3306, # port is 3306 by default
                 user = "username",
                 password = "password")
```

List and Import Tables

```
# lists the tables in the database  
dbListTables(con)  
# returns a dataframe  
dbReadTable(con, "tablename")  
# import all the tables  
table_names <- dbListTables(con)  
tables <- lapply(table_names, dbReadTable, conn = con)  
# close the connection when you are done  
dbDisconnect(con)
```

Importing Data From the Web

JSON

- ▶ JSON object: an unordered collection of name-value pairs.
- ▶ JSON array: an ordered sequence of zero or more values.
- ▶ JSON objects and arrays can be nested in each other.
- ▶ R handles JSON with the `jsonlite` package.

An Example with 'jsonlite'

- ▶ Let's get the current wind and delay status at LAX.

```
library(jsonlite)
airportCode <- "LAX"
url <- paste0("http://services.faa.gov/airport/status/",
              airportCode)
LAX <- fromJSON(url)
LAX$weather$wind
```

```
## [1] "Variable at 3.5mph"
```

```
LAX$delay
```

```
## [1] "false"
```

Quandl

- ▶ Quandl is a useful source of financial data and there is an R package Quandl to import the data into R.
- ▶ See <https://www.quandl.com/tools/r>.
- ▶ Data can be downloaded as xts objects, datatables, etc.

```
library(Quandl)
# download GDP as an xts object
gdp <- Quandl("FRED/GDP", type="xts")
last(gdp, 2)
```


Importing Data from WRDS

WRDS, CRSP, and R

- ▶ Wharton Research Data Services has over 250 terabytes of data.
- ▶ One data provider is The Center for Research in Security Prices (CRSP).
 - ▶ You will use CRSP data throughout the MFE program.
- ▶ I will show you how to access WRDS from R.
- ▶ Documentation: [Using R with WRDS](#)

Setup

- ▶ First, we need to obtain access to WRDS and download the SAS drivers for JDBC from [here](#).
- ▶ The two files should be saved locally.
- ▶ Take note of the path to the files; we need the path to establish the connection to WRDS.

Establish the Connection

```
# ---- INPUTS ---- #
username <- "myUserName"
password <- "myPassword"
# local path to the sas files
sasPath <- "C:/Users/myUser/Documents/wrds-drivers"
# ---- CODE ---- #
library(rJava)
options(java.parameters = '-Xmx4g')
library(RJDBC)
sasCore <- paste0(sasPath, "/sas.core.jar")
sasDriver <- paste0(sasPath, "/sas.intrnet.javatools.jar")
.jaddClassPath(c(sasCore, sasDriver))
driver <- RJDBC::JDBC(
  "com.sas.net.sharenet.ShareNetDriver",
  sasDriver, identifier.quote = "`")
wrds <- RJDBC::dbConnect(driver,
  "jdbc:sharenet://wrds-cloud.wharton.upenn.edu:8551/",
  username, password)
```

Accessing Data

- ▶ On the previous slide, we created the connection `wrds`.

```
res <- dbSendQuery(wrds, "select * from DATASET")
data <- fetch(res, n = -1)
data
```

- ▶ `dbSendQuery()` uses `wrds` to submit the SQL query string to WRDS, which then returns the result `res`.
- ▶ `select * from DATASET` is a SAS SQL query.
 - ▶ See the [SAS SQL Documentation](#) for more information.
- ▶ `fetch()` fetches the actual data based on the result `res`.
- ▶ `n = -1` is a parameter that determines how many observations to download.
 - ▶ `n = -1` specifies that we'd like unlimited observations returned.
 - ▶ `n = 10` would limit the number of observations returned to 10.

Example: S&P 500 Returns

```
sql <- "SELECT caldt, vwretd FROM CRSPQ.MSP500"
res <- dbSendQuery(wrds, sql)
dbHasCompleted(res) #check that this is true
msp500 <- fetch(res, n = -1)
dbClearResult(res) # free up memory
msp500$caldt <- as.Date(msp500$caldt)
library(xts)
msp500 <- xts::xts(msp500[, -1],
                   order.by = msp500$caldt)
colnames(msp500) <- "vwretd"
```

data.table

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 perfectly even when `data.frame` syntax is applied on `data.table`.
- ▶ `data.table` is very fast.
- ▶ The syntax of `data.table` is very concise.
 - ▶ Lowers programmer time...
 - ▶ ...but it can be hard to understand
 - ▶ Make sure you comment your code!

```
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
```

Types in R

- ▶ Character columns are very efficient in R.

```
typeof(1)
```

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

```
typeof(1L)
```

```
## [1] "integer"
```

```
typeof(NA)
```

```
## [1] "logical"
```

```
typeof(NA_integer_)
```

```
## [1] "integer"
```

Creating a data.table

```
DT <- data.table(A=1:6, B=c("a", "b", "c"),  
                  C=runif(6), D=FALSE)
```

DT

```
##      A B          C      D  
## 1:  1 a 0.5478945 FALSE  
## 2:  2 b 0.7732493 FALSE  
## 3:  3 c 0.1117793 FALSE  
## 4:  4 a 0.2044509 FALSE  
## 5:  5 b 0.8802859 FALSE  
## 6:  6 c 0.1667528 FALSE
```

Selecting Rows by Number in i

- ▶ The comma is optional.

```
DT[2:4, ]
```

```
##      A B      C      D
## 1: 2 b 0.7732493 FALSE
## 2: 3 c 0.1117793 FALSE
## 3: 4 a 0.2044509 FALSE
```

```
DT[2:4]
```

```
##      A B      C      D
## 1: 2 b 0.7732493 FALSE
## 2: 3 c 0.1117793 FALSE
## 3: 4 a 0.2044509 FALSE
```

Selecting Columns in j

- ▶ Columns are specified as a list with the actual names, not as character vectors.
- ▶ `.()` is an alias to `list()` in `data.tables`.

```
DT[2:3, list(A, C)]
```

```
##      A      C
## 1: 2 0.7732493
## 2: 3 0.1117793
```

```
DT[2:3, .(A, C)]
```

```
##      A      C
## 1: 2 0.7732493
## 2: 3 0.1117793
```

Selecting Columns in `j` with character vectors

- ▶ To select columns with a character vector, set the `with` argument to `FALSE`.

```
DT[2:3, c("A", "C"), with=FALSE]
```

```
##      A      C
## 1: 2 0.7732493
## 2: 3 0.1117793
```

Computing on Columns

- If the lengths of the results are not equal, the shorter one will be recycled.

```
DT[, .(Total = sum(A), Mean = mean(C))]
```

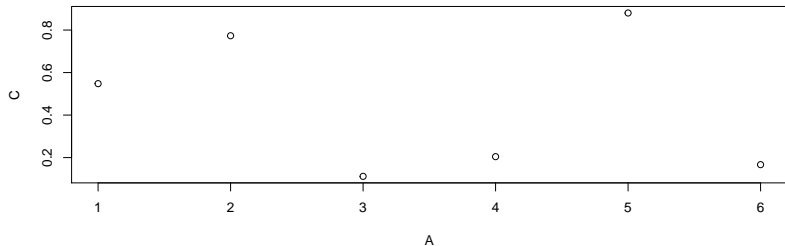
```
##      Total      Mean  
## 1:      21 0.4474021
```

```
DT[2:4, .(B, Mean = mean(C))]
```

```
##      B      Mean  
## 1: b 0.3631598  
## 2: c 0.3631598  
## 3: a 0.3631598
```

You can put almost anything into `j`

```
DT[, plot(A, C)]
```

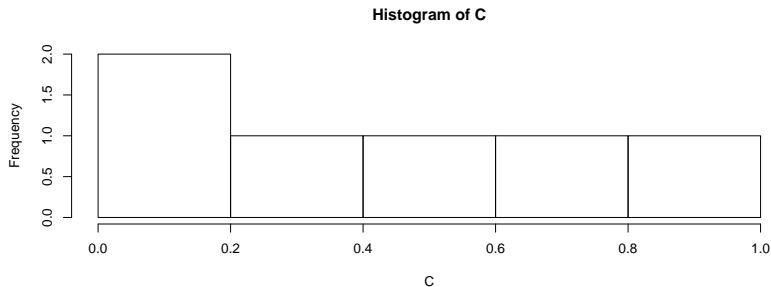


```
## NULL
```


Multiple Expressions Go in Curly Braces

```
DT[, {print(A)  
      hist(C)  
      NULL}] # set return value to NULL
```

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



```
## NULL
```

Returning a Vector.

```
DT[, .(A)]  # a data.table
```

```
##      A  
## 1: 1  
## 2: 2  
## 3: 3  
## 4: 4  
## 5: 5  
## 6: 6
```

```
DT[, A]  # a vector
```

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

Doing j by Group

```
DT[, .(Total = sum(A),  
      Mean = mean(C)),  
     by = .(B)] # returned in the order they appear
```

```
##      B Total      Mean  
## 1: a      5 0.3761727  
## 2: b      7 0.8267676  
## 3: c      9 0.1392660
```

```
# functions work as well  
DT[, .(Total = sum(C)), by = .(Group = A%%2)]
```

```
##      Group      Total  
## 1:      1 1.539960  
## 2:      0 1.144453
```

.N

- ▶ .N, when used inside square brackets, contains the number of rows.
- ▶ When put in `j`, .N counts the observations in each group.

```
DT[.N] # the last row, the same as DT[nrow(DT)]
```

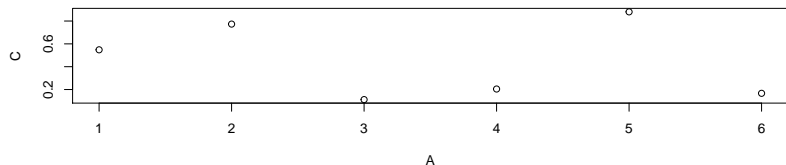
```
##      A B      C      D
## 1: 6 c 0.1667528 FALSE
```

```
DT[, .(Total = sum(C), Count = .N), by = .(Group = A%%2)]
```

```
##      Group      Total Count
## 1:      1 1.539960      3
## 2:      0 1.144453      3
```

Operations can be Chained Together

```
DT[, {plot(A, C)
      DT}][, .(Total = sum(C), # DT returns DT
              Count = .N),
            by = .(Group = A%%2)]
```



##	Group	Total	Count
## 1:	1	1.539960	3
## 2:	0	1.144453	3

Subset of Data - .SD

- ▶ .SD is a data.table.
- ▶ .SD holds all the columns except for the one specified in **by**, and .SD is only accessible in **j**.
- ▶ This is very useful if you have a very wide data.table.
- ▶ .SDcols allows you to apply a function to a subset of the columns.

```
DT <- as.data.table(iris)
DT[, lapply(.SD, median), by = Species]
```

##	Species	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width
## 1:	setosa	5.0	3.4	1.50	
## 2:	versicolor	5.9	2.8	4.35	
## 3:	virginica	6.5	3.0	5.55	

- ▶ Since lapply returns a list, we don't need to wrap it in .().

Add or Update Columns by Reference Using :=

```
DT <- data.table(A=1:3, B=4:6)
```

```
DT
```

```
##      A B
```

```
## 1:  1 4
```

```
## 2:  2 5
```

```
## 3:  3 6
```

```
DT[, c("A", "C") := .(rev(A), 7:9)]
```

```
##      A B C
```

```
## 1:  3 4 7
```

```
## 2:  2 5 8
```

```
## 3:  1 6 9
```

Remove Columns Using :=

- We use NULL to remove columns.

```
DT[, D := 10:12]
```

```
##      A B C  D
## 1:  3 4 7 10
## 2:  2 5 8 11
## 3:  1 6 9 12
```

```
DT[, c("B", "C") := NULL]
```

```
##      A  D
## 1:  3 10
## 2:  2 11
## 3:  1 12
```

- Wrap the LHS in () if you pass a variable.

Combining := with i and by

```
DT <- data.table(A=1:6, B=c("a", "b", "c"), C=runif(6))  
DT[1:5, D := sum(C), by=B]
```

##	A	B	C	D
## 1:	1	a	0.537051849	0.5417750
## 2:	2	b	0.656416493	1.2861747
## 3:	3	c	0.316941832	0.3169418
## 4:	4	a	0.004723194	0.5417750
## 5:	5	b	0.629758254	1.2861747
## 6:	6	c	0.867639604	NA

Use set() in Loops.

- ▶ set() is a loopable, low-overhead version, of the := operator, but it cannot handle grouping.
- ▶ Syntax: set(DT, i, j, value).
- ▶ Instead of for (i in 1:6) DT[i, z := i+1] we can

```
DT <- data.table(A=7:12, B=7:12)
for (i in 1:6) set(DT, i, 2L, i+1)
DT
```

```
##      A B
## 1:   7 2
## 2:   8 3
## 3:   9 4
## 4:  10 5
## 5:  11 6
## 6:  12 7
```

setnames() to Change the Column Names

- ▶ `setnames(DT, "old", "new")` changes the column names by reference (no copies are being made).

```
setnames(DT, c("A", "B"), c("X", "Y"))
```

```
DT
```

```
##      X Y
## 1:   7 2
## 2:   8 3
## 3:   9 4
## 4:  10 5
## 5:  11 6
## 6:  12 7
```

setcolorder() Reorders the Columns by Reference

```
setcolorder(DT,c("Y", "X"))  
DT
```

```
##      Y  X  
## 1:  2  7  
## 2:  3  8  
## 3:  4  9  
## 4:  5 10  
## 5:  6 11  
## 6:  7 12
```

Using Column Names in i

```
DT <- data.table(A=1:6, B=c("a", "b", "c"),  
                  C=runif(6), D=FALSE)  
DT[B %in% c("a","b")]
```

```
##      A B      C      D  
## 1: 1 a 0.57348148 FALSE  
## 2: 2 b 0.10619268 FALSE  
## 3: 4 a 0.03998307 FALSE  
## 4: 5 b 0.50723414 FALSE
```

Regular Expressions

- ▶ Metacharacters allow you to match certain types of characters.
 - ▶ For example, `.` means any single character, `^` means “begins with”, and `$` means “ends with”.
- ▶ If you want to use any of the metacharacters as actual text, you need to use the `\` escape sequence.
- ▶ See `?gsub()` and `?grep()`.

```
iris <- as.data.table(iris)
# Change column names
setnames(iris, names(iris), gsub("^Sepal\\.", "", names(iris)))
# Remove columns
iris[, grep("^Petal", names(iris)) := NULL]
```

##		Length	Width	Species
##	1:	5.1	3.5	setosa
##	2:	4.9	3.0	setosa
##	3:	4.7	3.2	setosa
##	4:	4.6	3.1	setosa

Keys

- ▶ Setting a key sorts the table by the column specified.

```
DT <- data.table(A=c("c", "b", "a"),B=1:6)
setkey(DT, A)
DT
```

```
##      A B
## 1:  a 3
## 2:  a 6
## 3:  b 2
## 4:  b 5
## 5:  c 1
## 6:  c 4
```

Keys as Row Names

- Keys can be used like row names.

```
DT["a"]
```

```
##      A B  
## 1: a 3  
## 2: a 6
```

```
DT["a", mult = "first"]
```

```
##      A B  
## 1: a 3
```

```
DT["a", mult = "last"]
```

```
##      A B  
## 1: a 6
```


nomatch

- Keys can be used like row names.

```
DT[c("a", "d")]
```

```
##      A B  
## 1: a  3  
## 2: a  6  
## 3: d NA
```

```
DT[c("a", "d"), nomatch = 0]
```

```
##      A B  
## 1: a  3  
## 2: a  6
```

Multi-Column Keys

- Use `.()` to select rows.

```
DT <- data.table(A=c("c", "b", "a"), B=1:6, C=7:12)
setkey(DT, A, B)
DT[.("b")]
```

```
##      A B  C
## 1: b  2  8
## 2: b  5 11
```

```
DT[.("b", 5)]
```

```
##      A B  C
## 1: b  5 11
```

Converting xts objects to data.tables

```
library(xts)
x <- matrix(1:4, nrow=2, ncol=2)
idx <- seq(as.Date("2016-10-31"), length=2, by="months")
x_xts <- xts(x, order.by = idx)
x_xts
```

```
##           [,1] [,2]
## 2016-10-31     1     3
## 2016-12-01     2     4
```

```
colnames(x_xts) <- c("a", "b")
DT <- as.data.table(x_xts)
setkey(DT, index)
DT
```

```
##           index a b
## 1: 2016-10-31  1  3
## 2: 2016-12-01  2  4
```

Rolling Joins

- ▶ Rolling joins are useful for time-series data.
- ▶ See `rollends` in `?data.table`.

```
DT
```

```
##           index a b
## 1: 2016-10-31  1 3
## 2: 2016-12-01  2 4
```

```
dt <- as.Date("2016-11-15"); DT[.(dt)]
```

```
##           index  a  b
## 1: 2016-11-15 NA NA
```

```
DT[.(dt), roll=TRUE] # roll forward; try roll=-Inf.
```

```
##           index a b
## 1: 2016-11-15  1 3
```

Lab 3

Let's work on Lab 3.