

MFE R Programming Workshop

Week 1

Brett Dunn and Mahyar Kargar

Fall 2017

Overview

Goals

- ▶ Learn to program in R.
- ▶ 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 as a language

- ▶ R is object oriented.
 - ▶ Everything is an object and functions operate differently when passed different types of objects.
- ▶ R is functional.
 - ▶ Everything that happens in R is a function call.
 - ▶ You write fewer loops.
 - ▶ You write cleaner code.
- ▶ R is extendable.
 - ▶ Interfaces to other software are part of R.

R vs C++

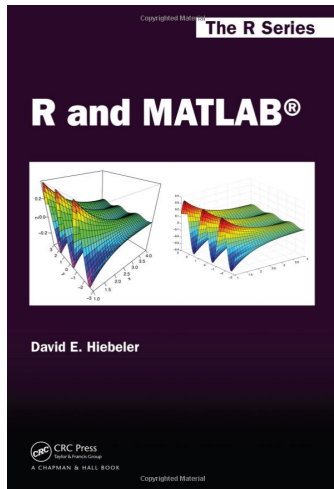
- ▶ Both are useful, and you will use both in the MFE program.
- ▶ R is an interpreted language.
 - ▶ Low programmer time.
 - ▶ A great tool for data munging, statistics, regressions, etc.
 - ▶ However, certain tasks in R can be slow (e.g. loops).
- ▶ C++ is very fast, but it takes longer to write programs.
- ▶ We can use both together!
- ▶ A good workflow:
 1. Write your program in R.
 2. If the program is too slow, benchmark your code.
 3. Try to speedup any bottlenecks in R.
 4. Convert any remaining bottlenecks to C++.

R vs MATLAB, Python, ect

- ▶ Each language has its own set of strengths and weakness.
- ▶ You are better served by learning R and C++ very well, rather than trying to learn R, C++, MATLAB, Python, Julia, SAS, etc.
- ▶ The MFE program is just too short.
 - ▶ You also need to learn finance!
- ▶ Once you are proficient with R and C++, learning other languages is easy.
- ▶ Don't become a master of none!

MATLAB

- ▶ If you want to learn MATLAB after learning R, take a look at [R and MATLAB](#) by David Hiebeler.



Structure

- ▶ I will talk at the beginning of each class.
- ▶ For the remainder of the time you will break into your study 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.
- ▶ 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 Resources: Books

- ▶ Introductory:
 - ▶ R for Everyone by Jared P. Lander
 - ▶ R Cookbook by Paul Teetor (free at [UCLA LearnIT](#))
 - ▶ R for Data Science by Hadley Wickham (free as well)
- ▶ Intermediate:
 - ▶ The Art of R Programming by Norman Matloff
- ▶ Advanced:
 - ▶ Software for Data Analysis by John Chambers
 - ▶ Extending R by John Chambers
 - ▶ Advanced R by Hadley Wickham

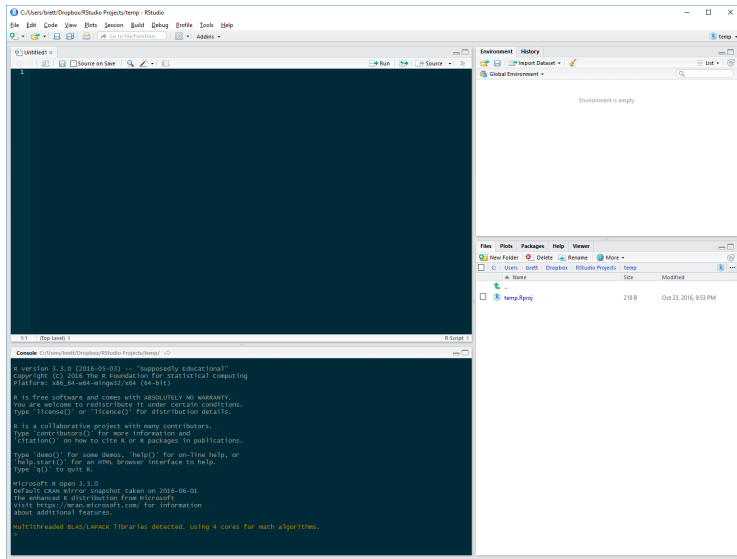
Other Resources

- ▶ Book series:
 - ▶ Use R! Springer series
 - ▶ FYI: Many Springer textbooks are just \$25 through <http://link.springer.com/>. You need to be on campus or signed into the UCLA VPN. You can download the pdfs for free.
 - ▶ O'Reilly R Books (free at [UCLA LearnIT](#))
- ▶ Built in documentation!
 - ▶ `?funcname`
- ▶ [Journal of Statistical Software](#)
- ▶ Data science courses on [Coursera](#)
- ▶ [Data Camp](#)
- ▶ <https://www.r-bloggers.com/>
- ▶ <https://twitter.com/rstudiotips>
- ▶ Google, Stack Overflow, etc.

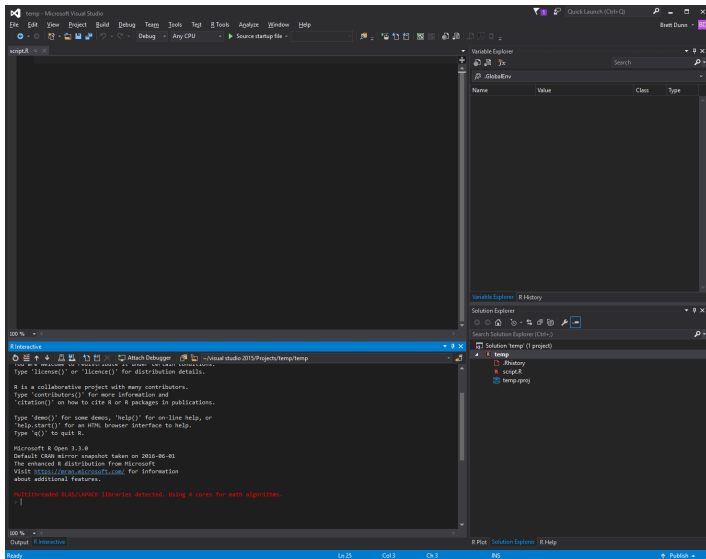
R Environment

- ▶ First, you need an R distribution.
 - ▶ I recommend Microsoft R Open.
 - ▶ <https://mran.microsoft.com/download/>
- ▶ Second, you need an integrated development environment (IDE) for R.
 - ▶ [R Studio](#) is a fantastic environment to interact with R.
 - ▶ Other options:
 - ▶ [R Tools for Visual Studio](#) if you use Visual Studio.
 - ▶ [Emacs Speaks Statistics \(ESS\)](#) if you use Emacs.
- ▶ 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.
- ▶ I will show you some Visual Studio.
- ▶ My focus is going to be on R programming.

RStudio

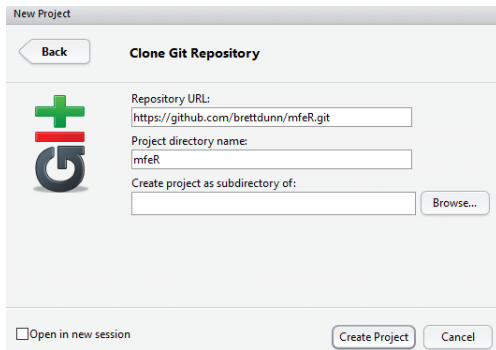


R Tools for Visual Studio



Course Materials

- ▶ <https://github.com/brettdunn/mfeR2017>
- ▶ The materials for this course were created in RStudio, using [R Markdown](#).
- ▶ To create your own RStudio project:
 - ▶ File / New Project / Version Control / Git
 - ▶ Enter the URL



R Basics

Command Line Interface

- ▶ To run a command in R, type it into the console next to the > symbol and press the Enter key.

```
2 + 3
```

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

- ▶ Up Arrow + Enter repeats the line of code.
- ▶ Esc (Windows/Mac) or Ctrl-C (Linux) interrupts a command.

RStudio

- ▶ To start, create a new R Script file.
 - ▶ File/New File/R Script
- ▶ You can type your commands in the R Script file and run them on the Console.
 - ▶ Easy way to save your work.
 - ▶ `Ctrl+Enter` sends the line at the cursor to the console.
 - ▶ `Ctrl+Shift+S` runs the entire file.
 - ▶ Help/Keyboard Shortcuts lists all the available shortcuts.
 - ▶ [RStudio Keyboard Shortcuts](#).
 - ▶ Check out the multiple cursors.
- ▶ For larger tasks with many files, create an R project.
- ▶ Visual Studio is similar.

General Comments

- ▶ Make your code easy to read.
- ▶ Check out [Google's R Style Guide](#)
- ▶ Comment your code!
 - ▶ # indicates a comment in R.
 - ▶ Ctrl+Shift+c comments the line.

Google's R Style Guide

R is a high-level programming language used primarily for statistical computing and graphics. The goal of the R Programming Style Guide is to make our R code easier to read, share, and verify. The rules below were designed in collaboration with the entire R user community at Google.

Summary: R Style Rules

1. [File Names](#): end in .R
2. [Identifiers](#): `variable.name` (or `variableName`), `FunctionName`, `kConstantName`
3. [Line Length](#): maximum 80 characters
4. [Indentation](#): two spaces, no tabs
5. [Spacing](#)
6. [Curly Braces](#): first on same line, last on own line
7. [else](#): Surround else with braces
8. [Assignment](#): use `<-`, not `=`
9. [Semicolons](#): don't use them
10. [General Layout and Ordering](#)
11. [Commenting Guidelines](#): all comments begin with # followed by a space; inline comments need two spaces before the #
12. [Function Definitions and Calls](#)
13. [Function Documentation](#)
14. [Example Function](#)
15. [TODO Style](#): `TODO(username)`

R Packages

- ▶ A package is essentially a library of prewritten code designed to accomplish some task or a collection of tasks.
- ▶ R has a huge collection of user-contributed packages.
 - ▶ Warning: Not all packages are of the same quality.



[CRAN](#)
[Mirrors](#)
[What's new?](#)
[Task Views](#)
[Search](#)

[About R](#)
[R Homepage](#)
[The R Journal](#)

[Software](#)
[R Sources](#)
[R Binaries](#)
[Packages](#)
[Other](#)

[Documentation](#)
[Manuals](#)
[FAQs](#)
[Contributed](#)

[Bayesian](#)
[ChemPhys](#)
[ClinicalTrials](#)
[Cluster](#)
[DifferentialEquations](#)
[Distributions](#)
[Econometrics](#)
[Environmetrics](#)
[ExperimentalDesign](#)
[ExtremeValueTheory](#)
[Finance](#)
[Genetics](#)
[Graphics](#)
[HighPerformanceComputing](#)
[MachineLearning](#)
[MedicalImaging](#)
[MetaAnalysis](#)
[Multivariate](#)
[NaturalLanguageProcessing](#)
[NumericalMathematics](#)
[OfficialStatistics](#)

CRAN Task Views

Bayesian Inference
Chemometrics and Computational Physics
Clinical Trial Design, Monitoring, and Analysis
Cluster Analysis & Finite Mixture Models
Differential Equations
Probability Distributions
Econometrics
Analysis of Ecological and Environmental Data
Design of Experiments (DoE) & Analysis of Experimental Data
Extreme Value Theory
Empirical Finance
Statistical Genetics
Graphic Displays & Dynamic Graphics & Graphic Devices & Visualization
High-Performance and Parallel Computing with R
Machine Learning & Statistical Learning
Medical Image Analysis
Meta-Analysis
Multivariate Statistics
Natural Language Processing
Numerical Mathematics
Official Statistics & Survey Methodology

R Packages

- ▶ Installing a packages:
 - ▶ Ctrl+7 in RStudio accesses the packages pane
 - ▶ You can also type `install.packages("packageName")`
- ▶ Uninstalling a package:
 - ▶ `remove.packages("packageName")`
- ▶ Loading packages:
 - ▶ `require(packageName)` or `library(packageName)` loads a package into R
 - ▶ The difference is that `require` returns `TRUE` if the package loads or `FALSE` if it doesn't.
- ▶ Unloading packages
 - ▶ `detach(package:packageName)`
- ▶ If two packages have the same function name use two colons:
`-package1::func` or `package2::func`

Getting Help in R

- ▶ To get help on a function, use `?`.
- ▶ The `example` function runs the examples contained in the help file.
- ▶ To run a search through R's documentation, use `??`.
- ▶ To get help on a package, type
`help(package="packageName")`

```
?seq    # pulls up the help page
example(seq)  # runs the examples in R
??"normal distribution"  # runs a search
help(package = "xts")  # gets help on the xts package
?'+'  # gets help on the + function
```

Variables

- ▶ Unlike C++, R does not require variable types to be declared.
- ▶ A variable can take on any data type.
- ▶ A variable can also hold any R object such as a function, the result of an analysis, a plot, etc.
- ▶ Variable assignment is done with `<-` (Alt+- in RStudio).
 - ▶ `=` works, but there are reasons to prefer `<-`.
- ▶ We can remove variables (e.g. to free up memory) with the `rm` function.
 - ▶ `gc()` runs garbage collection.
 - ▶ `rm(list=ls())` clears the workspace.

```
x <- 2  # x is a pointer  
x      # the same output as print(x)
```

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

```
rm(x)  # removes x
```

Data Types

- ▶ There are many different data types in R.
- ▶ The four main types of data most likely to be used are:
 1. numeric
 2. character (string)
 3. Date/POSIXct (time-based)
 4. logical (TRUE/FALSE)
- ▶ The data type can be checked with the `class` function

```
x <- as.Date("2010-12-21")  
class(x)
```

```
## [1] "Date"
```


Casting

```
x <- "2010-12-21"  
class(x)
```

```
## [1] "character"
```

```
x
```

```
## [1] "2010-12-21"
```

```
x <- as.Date(x)  
class(x)
```

```
## [1] "Date"
```

```
x
```

```
## [1] "2010-12-21"
```

More Casting

```
x <- as.numeric(x)  
class(x)
```

```
## [1] "numeric"
```

```
is.numeric(x)
```

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

```
x # number of days since Jan 1, 1970
```

```
## [1] 14964
```

Even More Casting

```
x <- as.integer(x)  # x <- 14964L assigns an integer  
class(x)
```

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

```
is.integer(x)
```

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

```
is.numeric(x)  # R promotes int to numeric as needed
```

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

```
4L / 5L
```

```
## [1] 0.8
```

Logicals

```
# TRUE == 1 and FALSE == 0  
x <- TRUE # TRUE, FALSE, T, F are logicals  
is.logical(x)
```

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

```
5 == 5 # != tests for inequality
```

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

```
"a" < "b" # works on characters as well
```

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

Vectors

Vectors

- ▶ A vector is a collection of elements, all of the *same* type.
- ▶ In R, a vector does not have a dimension attribute.
 - ▶ There is no difference between a row vector and a column vector.
- ▶ We will learn about:
 - ▶ Recycling
 - ▶ The automatic lengthening of vectors.
 - ▶ Filtering
 - ▶ The extraction of subsets of vectors.
 - ▶ Vectorization
 - ▶ Where functions are applied element-wise to vectors.

Vectors and Assignment

- ▶ Assigning values to variables can be done with `<-`.
- ▶ Often, we create vectors using the `c()` function.
 - ▶ The “c” stands for combine because the arguments into a vector.

```
x <- c(1, 2, 3, 4)
x
```

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

```
y <- c(x, 5, 6)
y
```

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

Creating Vectors with seq and rep

- ▶ Both seq and rep are useful functions for generating vectors.
- ▶ See ?seq and ?rep for details
- ▶ seq is also useful in loops
- ▶ 1:10 is the same as seq(1,10,1)

```
x <- seq(from = 1, to = 10, by = 2)
x
```

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

```
y <- rep(c(1, 2), times = 3)
y
```

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

```
rep(c(1,2), each=2)
```

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


Obtaining the Length of a Vector

- ▶ `length()` returns the vector length

```
x <- c(TRUE, FALSE, TRUE, FALSE)
length(x)
```

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

```
x <- c()      # x is NULL
1:length(x)   # that could mess you up in a for loop
```

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

```
seq(x)        # a safe way to loop through a vector
```

```
## integer(0)
```

Accessing Elements of Vectors

- ▶ Elements can be accessed using `[]`
 - ▶ Help on the `[]` function can be found by typing `?'[]'`
- ▶ Unlike C/C++, R indexing starts at 1, not 0.
- ▶ The `[]` function can take a vector as an arguments.

```
x <- c("a", "b", "c", "d")  
x[1]                                # access the first element
```

```
## [1] "a"
```

```
x[c(1, 3)]                          # access elements 1 and 3
```

```
## [1] "a" "c"
```

```
x[c(TRUE, FALSE, TRUE, FALSE)]    # second way
```

```
## [1] "a" "c"
```

NULL and NA

- ▶ NULL is the non-existent value in R.
- ▶ NA is the missing place holder.

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

```
## [1] 5 NA 7 8
```

```
y <- NULL  
length(y)
```

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

Names of Vector Elements

- ▶ You can give names to elements of vectors, and you can access elements by their name.
- ▶ The function `as.vector` removes the names from a vector.

```
x <- 1:3
names(x) <- c("A", "B", "C")
x <- c(A=1, B=2, C=3 ) # another way
x["B"]
```

```
## B
## 2
```

```
as.vector(x) # the names are removed
```

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

Recycling

- ▶ When applying an operation to two vectors that requires them to be the same length, R automatically *recycles* the shorter one, until it is long enough to match the longer one.
- ▶ Be careful with and aware of this behavior!
- ▶ In some cases it is useful, others confusing.

```
# the shorter vector will be recycled  
c(2, 4, 6) + c(1, 1, 1, 2, 2, 2)
```

```
## [1] 3 5 7 4 6 8
```

```
# this is the same as  
rep(c(2, 4, 6), 2) + c(1, 1, 1, 2, 2, 2)
```

```
## [1] 3 5 7 4 6 8
```

Logical Operators

- ▶ R has several logical operations that act on vectors.
- ▶ `!`, `==`, `!=`, `&`, `&&`, `|`, `||`, `xor()`, `any()`, `all()`, `>`, `>=`, `<=`, `<`

```
x <- c(TRUE,FALSE,TRUE)
y <- c(TRUE,FALSE,FALSE)
x == y
```

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

```
!x
```

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

Logical Operations (2)

- ▶ `&&`, `||`, `any()`, and `all()` return a length-one vector.
- ▶ The shorter forms are vectorized, meaning they can return a vector:

```
x <- c(FALSE, TRUE, TRUE)
y <- c(TRUE, TRUE, FALSE)
x & y
```

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

- ▶ The longer form evaluates left to right examining only the first element of each vector:

```
x && y
```

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

Filtering

- We select subsets of vectors with vectors of logicals.

```
x <- 1:5  
y <- c(TRUE,FALSE,TRUE,FALSE,TRUE)  
x[y]
```

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


Filtering (2)

- ▶ Filtering amounts to generating filtering indices (i.e. vectors of logicals).

```
x <- c(5, 2, -3, 8)
idx <- x*x > 8
idx
```

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

```
# another way
">"(x*x, 8)
```

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

Assigning to a Filter

- ▶ You can assign elements to the subsets.
 - ▶ This allows you change elements that meet certain criteria.

```
x <- 1:6  
x[x <= 2] <- NA  
x
```

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

Filtering with subset()

- ▶ The subset function filters and removes any NAs.

```
x <- c(3, 1:5, NA, 79)
x
```

```
## [1] 3 1 2 3 4 5 NA 79
```

```
x[x > 4]
```

```
## [1] 5 NA 79
```

```
subset(x, x > 4)
```

```
## [1] 5 79
```

Filtering with Indices

- ▶ Select using the row/column number.
- ▶ Exclude entries with a negative vector.

```
x <- 10:20  
x[c(1:2, 6:7)]
```

```
## [1] 10 11 15 16
```

```
x[-c(1,3)] # removes 1st and 3rd obs
```

```
## [1] 11 13 14 15 16 17 18 19 20
```

The Selection Function `which()`

- ▶ `which()` gives us the *position* in a vector where a condition occurs.

```
x <- c(3, 1:5, NA, 79)
x > 4
```

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

```
which(x > 4)
```

```
## [1] 6 8
```

- ▶ See <https://stackoverflow.com/questions/6918657/whats-the-use-of-which>.

%in%

- ▶ %in% returns a logical vector indicating if there is a match or not for its left operand.

```
1:5 %in% c(1,3)
```

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

Vectorization: Functions on Vectors

- ▶ R functions typically operate on vectors.
- ▶ Often, there is an argument to ignore missing data.

```
x <- c(1:1000, NA)
mean(x)
```

```
## [1] NA
```

```
mean(x, na.rm = TRUE)
```

```
## [1] 500.5
```

```
log(x)[998:1001]
```

```
## [1] 6.905753 6.906755 6.907755      NA
```

Lists

Creating Lists

- ▶ A list is a structure that combines objects of *different* type and length.
- ▶ You can create a list where the elements are of type list.

```
element1 <- 1:5  
element2 <- matrix(1:6, nrow=2)  
mylist <- list(el1=element1, el2=element2)  
mylist
```

```
## $el1  
## [1] 1 2 3 4 5  
##  
## $el2  
##      [,1] [,2] [,3]  
## [1,]    1    3    5  
## [2,]    2    4    6
```

Accessing Elements of Lists

- We can access a list component in several different ways.

```
mylist <- list(A=1, univ=c("UCLA", "USC"),  
              mymat=matrix(1:4, nrow=2))  
mylist[[1]]      # first way
```

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

```
mylist[["A"]]    # second way
```

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

```
mylist$A         # third way
```

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

Removing Components of Lists

- ▶ We can delete a component of a list by setting it to NULL.

```
mylist <- list(A=1)
mylist$B <- c(1, 2)  # adds a component to a list
mylist
```

```
## $A
## [1] 1
##
## $B
## [1] 1 2
```

```
mylist$A <- NULL
mylist
```

```
## $B
## [1] 1 2
```

Subsetting Lists

- ▶ Subsets of lists are done with single `[]`.
- ▶ A single `[]` returns a sublist of the original list

```
mylist <- list(A=1, univ=c("UCLA", "USC"),  
              mymat=matrix(1:4, nrow=2))  
# this returns a list because of the single []  
mylist[c(1,3)]
```

```
## $A  
## [1] 1  
##  
## $mymat  
##      [,1] [,2]  
## [1,]    1    3  
## [2,]    2    4
```

Lists of Objects

- Lists can store all types of objects.

```
l <- list(a = matrix(1:4, nrow=2),  
          b = list(A=1:10,B=20:30))  
l[["a"]]
```

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

```
l[[2]][[1]]
```

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

Simplifying vs Preserving Subsetting

- ▶ Note that `[[` simplifies the result, returning a vector (`str` displays the structure of an R object):

```
str(mylist[[2]])
```

```
## chr [1:2] "UCLA" "USC"
```

- ▶ But `[` is preserving subsetting and returns a list:

```
str(mylist[2])
```

```
## List of 1
```

```
## $ univ: chr [1:2] "UCLA" "USC"
```

Applying Functions to a List with lapply

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

```
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
```

An Example of lapply

- From ?lapply: lapply(X, FUN, ...) returns a list of the same length as X, each element of which is the result of applying FUN to the corresponding element of X.

```
l <- c("A","B","B","A","A","B")  
lapply(c("A","B"), function(letter) which(l==letter))
```

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


Investments Problem Set 1

- You will submit a list of answers.

```
hw <- source("../lecture1p.R")$value  
str(hw)
```

```
## List of 6  
## $ student: chr [1:4] "Molin Liang" "Meghana Rao" "Cheng  
## $ Q1      : num [1:3] 1 2 3  
## $ Q2      : num [1:2] 1 2  
## $ Q3      : num [1:2] 1 2  
## $ Q4      : num [1:3] 1 2 3  
## $ Q5      :List of 2  
## ..$ a: num [1:5] 0 0 0 0 0  
## ..$ b: num [1:5] 0 0 0 0 0
```

Functions

Function Definitions

- ▶ The last value evaluated is what is returned by the function.
- ▶ You can also write `return(x^2)`.
 - ▶ This is useful if you want to break out of the function early.

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

```
## [1] 100
```

Scope Rules for Functions

- ▶ Variables defined inside a function are local to that function.

```
myfunc <- function(x) {  
  N <- 10  
  return(N*x^2)  # return is optional  
}  
myfunc(10)
```

```
## [1] 1000
```

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

$\%>\%$

The Pipe Operator %>%

- ▶ The `magrittr` package provides a pipe operator.
- ▶ See `vignette("magrittr")`.
- ▶ Basic piping:
 - ▶ `x %>% f` is equivalent to `f(x)`
 - ▶ `x %>% f(y)` is equivalent to `f(x, y)`
 - ▶ `x %>% f %>% g %>% h` is equivalent to `h(g(f(x)))`
- ▶ The argument placeholder:
 - ▶ `x %>% f(y, .)` is equivalent to `f(y, x)`
 - ▶ `x %>% f(y, z = .)` is equivalent to `f(y, z = x)`

Expose the variables with %\$%

- ▶ The %\$% allows variable names (e.g. column names) to be used in a function.

```
library(magrittr)
iris %>%
  subset(Sepal.Length > mean(Sepal.Length)) %$%
  cor(Sepal.Length, Sepal.Width)
```

```
## [1] 0.3361992
```

Compound assignment pipe operations with %<>%

- ▶ There is also a pipe operator which can be used as shorthand notation in situations where the left-hand side is being “overwritten”:

```
iris$Sepal.Length <-  
  iris$Sepal.Length %>%  
  sqrt()
```

Use the %<>% operator to avoid the repetition:

```
iris$Sepal.Length %<>% sqrt
```

- ▶ This operator works exactly like %>%, except the pipeline assigns the result rather than returning it.

Lab 1

Let's work on Lab 1.