MFE R Programming Workshop

Week 1

Brett R. Dunn

Fall 2016

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++

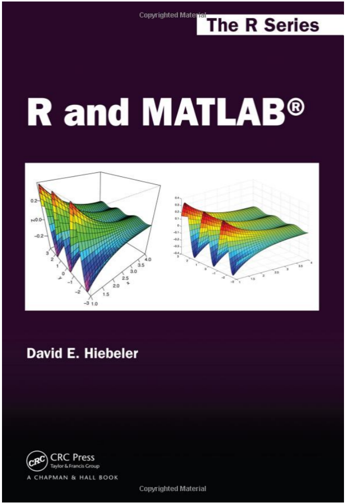
- Both are useful, and you will use both in the MFE program.
- R is an interpreted language.
  - Low programmer time.
  - A great tool for data munging, statistics, regressions, etc.
  - However, certain tasks in R can be slow (e.g. loops).
- C++ is very fast, but it takes longer to write programs.
- 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++.

Jack of All Trades, Master of None

- 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:
  - 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 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

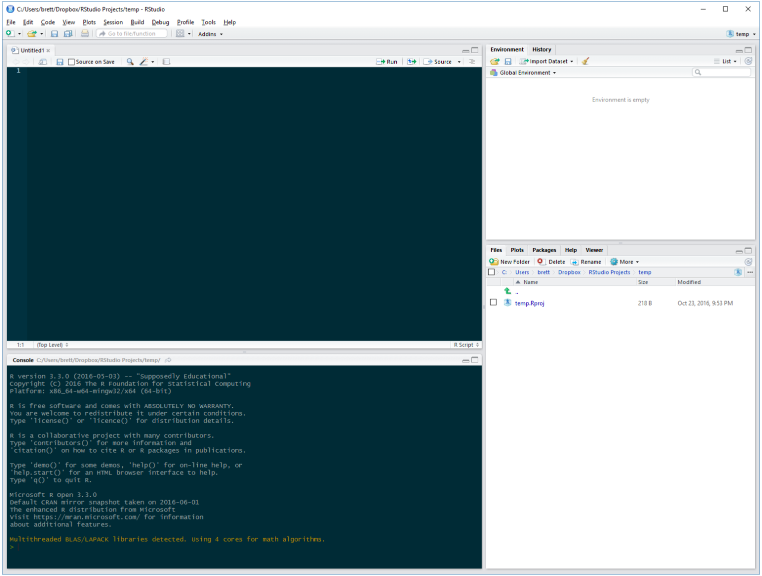
Other Resources

- 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

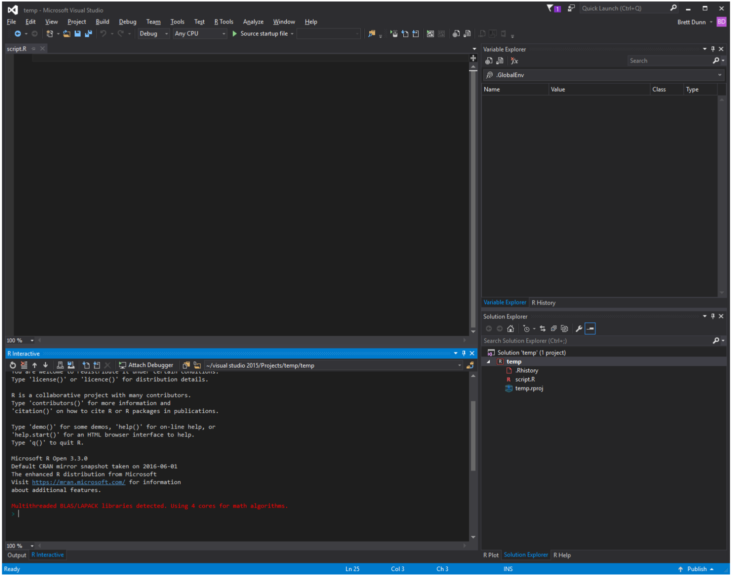
- First, you need an R distribution.
  - I recommend Microsoft R Open.
  - https://mran.revolutionanalytics.com/download/
- Second, you need an integrated development environment (IDE) for R.
  - R Studio is a fantastic environment to interact with R.
  - Other options:
    - R Tools for Visual Studio 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



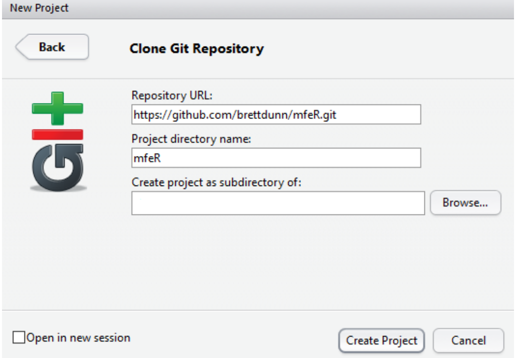
C:/Users/brett/Dropbox/RStudio Projects/temp - RStudio
File Edit Code View Plots Session Build Debug Profile Tools Help
Go to file/function
Addins
temp
Untitled1
Source on Save
Run
Source
1:1
(Top Level)
R Script
Console C:/Users/brett/Dropbox/RStudio Projects/temp/
R version 3.3.0 (2016-05-03) -- "Supposedly Educational"
Copyright (C) 2016 The R Foundation for Statistical Computing
Platform: x86_64-w64-mingw32/x64 (64-bit)
R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.
R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.
Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.
Microsoft R Open 3.3.0
Default CRAN mirror snapshot taken on 2016-06-01
The enhanced R distribution from Microsoft
Visit https://mran.microsoft.com/ for information
about additional features.
Multithreaded BLAS/LAPACK libraries detected. Using 4 cores for math algorithms.
Environment
History
Import Dataset
List
Global Environment
Environment is empty
Files
Plots
Packages
Help
Viewer
New Folder
Delete
Rename
More
C: Users brett Dropbox RStudio Projects temp
Name
Size
Modified
temp.Rproj
218 B
Oct 23, 2016, 9:53 PM

R Tools for Visual Studio



Course Materials

- https://github.com/brettdunn/mfeR
- 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.

```r
2 + 3
```

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

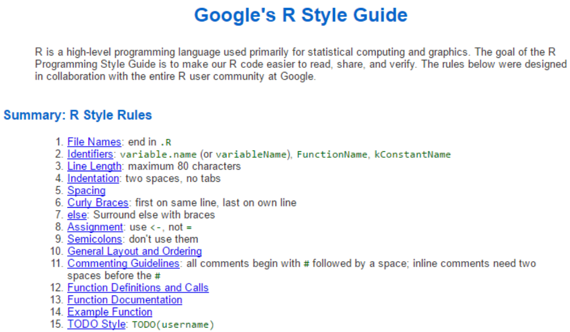
- Up Arrow + Enter repeats the line of code.
- Esc (Windows/Mac) or Ctrl-C (Linux) interrupts a command.

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.
    - 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.



Google's R Style Guide

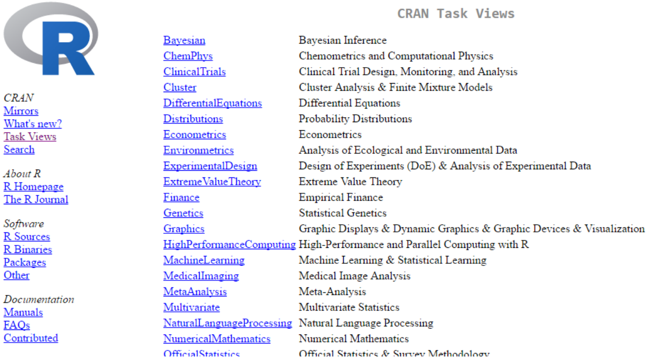
R is a high-level programming language used primarily for statistical computing and graphics. The goal of the R Programming Style Guide is to make our R code easier to read, share, and verify. The rules below were designed in collaboration with the entire R user community at Google.

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

CRAN Task Views

Bayesian	Bayesian Inference
ChemPhys	Chemometrics and Computational Physics
ClinicalTrials	Clinical Trial Design, Monitoring, and Analysis
Cluster	Cluster Analysis & Finite Mixture Models
DifferentialEquations	Differential Equations
Distributions	Probability Distributions
Econometrics	Econometrics
Environmetrics	Analysis of Ecological and Environmental Data
ExperimentalDesign	Design of Experiments (DoE) & Analysis of Experimental Data
ExtremeValueTheory	Extreme Value Theory
Finance	Empirical Finance
Genetics	Statistical Genetics
Graphics	Graphic Displays & Dynamic Graphics & Graphic Devices & Visualization
HighPerformanceComputing	High-Performance and Parallel Computing with R
MachineLearning	Machine Learning & Statistical Learning
MedicalImaging	Medical Image Analysis
MetaAnalysis	Meta-Analysis
Multivariate	Multivariate Statistics
NaturalLanguageProcessing	Natural Language Processing
NumericalMathematics	Numerical Mathematics
OfficialStatistics	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")`

```r
?seq  # pulls up the help page
example(seq)  # runs the examples in R
??"normal distibution"  # 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 `<-`.
  - = 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.

```
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

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

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

```r
x
```

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

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

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

```r
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.

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

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

```r
# 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.

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

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

```
x & y
```

```
## [1]  TRUE FALSE 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  # same as ">"(x*x, 8) - 8 is recycled!
idx
```

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

Filtering with `subset()`

- The subset function filters and removes any `NA`s.

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

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

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

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

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

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

The Selection Function `which()`

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

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

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

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

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

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

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

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
```

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, byrow = FALSE)
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, byrow = FALSE)
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, byrow = FALSE)
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
```

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
```

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!

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

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

```
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
```

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]]  # fist 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
```

Applying Functions to a List with `lapply`

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

```r
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
```

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, a `data.frame` is different because each column can have a different mode.
  - For example, one column might be numbers and another characters.
- 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`.

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

- 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)

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

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

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

```r
gradeBook[1,]
```

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

```r
gradeBook[,2]
```

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

Merging `data.frames`

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

```r
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

```
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.

```
optdata <- read.csv(file="./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
```

Long example

```
all2006 <- read.csv("2006.csv",header=TRUE,as.is=TRUE)

# exclude hourly-wagers
all2006 <- all2006[all2006$Wage_Per=="Year", ]

# exclude weird cases
all2006 <- all2006[all2006$Wage_Offered_From > 20000,]

all2006$rat <- all2006$Wage_Offered_From
                  / all2006$Prevailing_Wage_Amount

se2006 <- all2006[grep("Software Engineer", all2006),]
```

Control Statements

For loops (1)

- A `for` loop iterates over an index, provided as a vector.
- To iterate over the length of a vector `x`, we can either use `1:length(x)` or `seq(x)`.
  - `seq(x)` protects against zero-length vectors.

```
x <- c(1:5)
y <- NULL  # we need to initialize an empty vector
for(i in seq(x)) {  # safer than 1:length(x)
   y[i] <- x[i] + 2
}
y
```

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

For loops (2)

- Another nice way to make a `for` loop.

```
x <- c(2:4)
for(i in x) {
   print(i + 2)
}
```

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

While loops

- A `while` loop runs the code inside the braces repeatedly as long as the tested condition proves `TRUE`.

```
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
```

Intro to Conditional Statements

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

```
## [1] 10
```

Functions

Function Definitions

- Note that the last value evaluated is what is returned by the function.
- You can also write `return(x^2)`.
  - I prefer this because the code is clearer.

```
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
```

Lab 1

Let's work on Lab 1.