

# MFE R Programming Workshop

## Week 2

Brett Dunn and Mahyar Kargar

Fall 2017

# Introduction

# Questions

Any questions before we start?

# Overview of Week 2

- ▶ Functions
- ▶ Optimization
- ▶ Control Statements
- ▶ Lab

# Functions

# Everything that happens in R is a function call

```
`<-`(mynumber, 3)  
print(mynumber)
```

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

```
`+`(3,4)
```

```
## [1] 7
```

```
a <- `:`(11,20)  
`[`(a,5)
```

```
## [1] 15
```

# Function Definitions

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

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

- ▶ 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.

# Scope Rules for Functions

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

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

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

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



## Default Arguments

- ▶ R makes frequent use of default arguments.
- ▶ Default arguments are initialized with a default value unless the user specifies an override.

```
g <- function(x, y=2, z=2) x*y*z  
g(2)
```

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

```
g(2, 1)
```

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

```
g(2, 1, 1)
```

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

$\%>\%$

# The Pipe Operator %>%

- ▶ Pipes are a powerful tool for clearly expressing a sequence of multiple operations.
- ▶ 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)`

## An Example of %>%

```
library(magrittr)
x <- 1:10
x %>% mean
```

```
## [1] 5.5
```

```
y <- c(x, NA)
y %>% mean(na.rm=TRUE)
```

```
## [1] 5.5
```

## Expose the variables with %\$%

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

```
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.

# Control Statements

# WARNING!

- ▶ In general, loops are *slow* in R.
- ▶ Vectorized code is faster.
- ▶ Loops are often a good place to write a R function in C++.
  - ▶ We will cover this later.



# Looping vs Vectorization

```
library(microbenchmark)
sq <- function(x) {
  n <- length(x)
  res <- rep(NA, n)
  for(i in 1:n) res[i] <- x[i]^2
  res
}
x <- 1:1000
microbenchmark(
  sq(x),
  x^2
)
```

## Unit: microseconds

##	expr	min	lq	mean	median	uq	max	neval
##	sq(x)	201.077	202.008	293.58463	203.405	208.214	9049.380	100
##	x^2	5.585	5.896	6.13815	5.896	6.206	14.274	100

## 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
```

# If Statements

- ▶ If statements operate on length-one logical vectors.
- ▶ Syntax:
  - ▶ `if(cond1=true) { cmd1 } else { cmd2 }`

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

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

# Ifelse Statements

- ▶ Ifelse statements operate on vectors of variable length.
- ▶ Syntax:
  - ▶ `ifelse(test, true_value, false_value)`

```
x <- 1:10 # Creates sample data  
ifelse(x<5 | x>8, x, 0)
```

```
## [1] 1 2 3 4 0 0 0 0 9 10
```

## Optimization in R

# Overivew

- ▶ R has many tools to solve optimization problems.
- ▶ See [CRAN Task View: Optimization and Mathematical Programming](#).
- ▶ [Continuous Global Optimization in R](#)



# One Dimensional Root (Zero) Finding

- ▶ The function `uniroot` searches an interval for a root (i.e., zero) of the function `f` with respect to its first argument.
- ▶ Suppose we want  $x$  such that  $5 - e^{x^2} = x^2$ .
- ▶ We convert this to the root finding problem  $5 - e^{x^2} - x^2 = 0$ .

```
f <- function(x) {  
  5 - exp(x^2) - x^2  
}  
root <- uniroot(f, c(0, 2), tol = 10^-8)  
root$root
```

```
## [1] 1.143048
```

```
f(root$root)
```

```
## [1] 9.738421e-09
```

# General-Purpose Optimization with `optim`

## ► Rosenbrock Banana function

```
fr <- function(x) {    ## Rosenbrock Banana function
  x1 <- x[1]
  x2 <- x[2]
  100 * (x2 - x1 * x1)^2 + (1 - x1)^2
}
optim(c(-1.2,1), fr)$par
```

```
## [1] 1.000260 1.000506
```

## Investments Problem Set 2

- ▶ You will use R to calculate yield-to-maturity.

# Lab 1

# Reading in Data for Lab 1

- ▶ `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`.

```
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
```

- ▶ `optdata` is a `data.frame`, a specialized type of `list`
- ▶ `write.csv` writes data to a `.csv` file.

# Lab 1

Let's work on Lab 1.