Functional Programming in R

Functions in R

- Functions in R are treated just like any other data.
- They are assigned to variables just like any other object.
- They can be passed to other functions as arguments and returned by other functions as output.
- They can also be anonymous, meaning they can be passed as an argument with out being assigned to a variable.

for loops

You are probably used to `for` loops:

```
myData <- c(1, 2, 3, 4, 5)
for(i in 1:length(myData)){
    myData[i] <- myData[i] + 1
}
myData
```

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

Functional Programming Basics in R

Alternatives to for loops when working with R objects.

- Vectorized Operations
- Functionals (High Order Functions)

These abstract away the looping construct and pass those instructions to often highly optimized code that does the looping for the user.

R objects are immutable (usually)

> *Internally, `x$a <- 2` is a combination of two steps: first construct a modified copy of `x`, then change the binding of `x` to the new object; the bindings change, but the original `x` does not. This makes R objects immutable: whenever it looks like you are modifying an object, you are actually creating a modified copy.*
>
> *~ Mutable Objects in R, Hadley Wickham 2010*

This means that objects in R can't usually be changed in place. For large objects, this can mean copying big pieces of memory to change small bits of an object.

This is not true for all R objects but its a good rule of thumb to go by unless you know otherwise.

Vectorized Operations

Vectorized operations simplify common looping tasks and they are found everywhere in R.

Create a second column that is one more than the first.

```
myData <- 1:15
for(i in 1:length(myData)){
    myData[i] <-  myData[i] + 1
}
```

This is a vectorized version of the same task.

```
myData <- 1:15
myData <- myData + 1
```

Vector rule of recycling

Vectors of short lengths are recycled. Warnings are thrown when the shorter vector is not a factor of the larger.

```
a <- 1:10
b <- c(2, 10)
a * b
```

```
##  [1]   2  20   6  40  10  60  14  80  18 100
```

```
a <- 1:10
b <- 1:3
a * b
```

```
## Warning in a * b: longer object length is not a multiple of shorter
## length
```

```
##  [1]  1  4  9  4 10 18  7 16 27 10
```

Functionals

A function that takes an object and applys a function to each element of that object.

The function that is applied is passed as an argument.

In R these are commonly the *apply family of functions.

- apply - runs on the margins of 2D data
- lapply - returns a list
- sapply - returns a vector, if possible
- mapply - multivariable lapply

Example

- Assembles a vector of randomly uniform values,
- define a function that will check a value,
- then apply the function to the vector with sapply.

```
ckFun <- function(x){
    if(x > 50) return(x - 50)
    return(x)
}
x <- round(runif(1000, 1, 100))
print(head(x, 10))
```

```
##  [1] 27 38 58 91 21 90 95 66 63  7
```

```
newx <- sapply(x, ckFun)
print(head(newx, 10))
```

```
##  [1] 27 38  8 41 21 40 45 16 13  7
```

A vectorized approach with `ifelse`

You could also use the `ifelse` function as well.

This handy function checks if a statement is `TRUE` or `FALSE` for every element of a vector then returns a result depending on the condition.

This would return the same results as the process on the previous slide.

```
xnew <- ifelse(x > 50, x - 50, x)
```

Benchmarking the different approaches

We will compare the performance of these 3 approaches on a single vector and a column in a data frame.

```
library(microbenchmark)

vecLen <- 1000
vec <- round(runif(vecLen, 1, 100))
dat <- data.frame(col1 = vec)
```

for loops

```
forV <- function(vec){
    for(i in 1:length(vec)){
        if(vec[i] > 50){
            vec[i] <- vec[i] - 50
        }
    }
    return(vec)
}

forD <- function(dat){
    for(i in 1:nrow(dat)){
        if(dat[i,1] > 50){
            dat[i,2] <- dat[i,1] - 50
        }
    }
    return(dat)
}
```

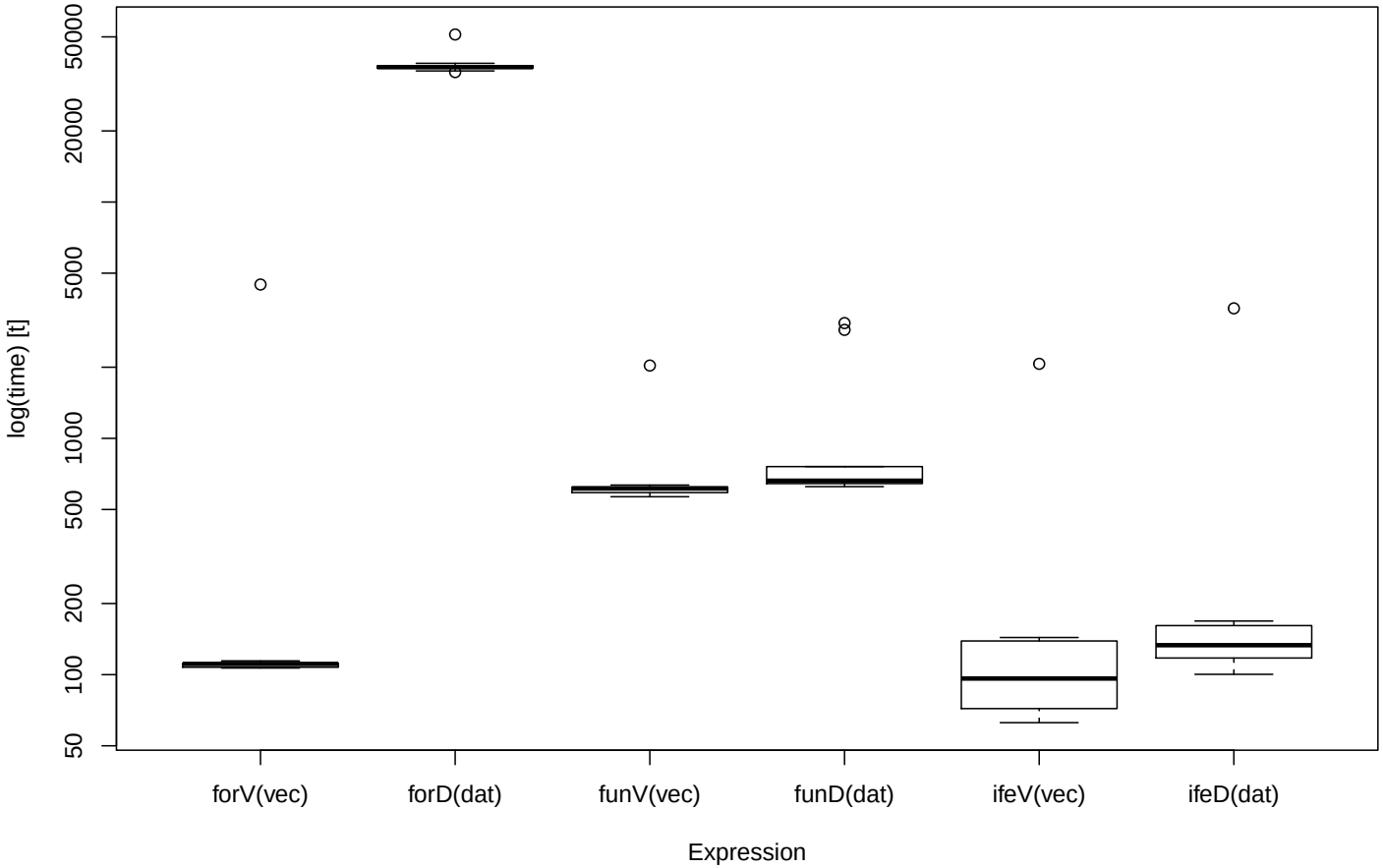
sapply

```
ckFun <- function(x){
    if(x > 50){
        return(x - 50)
    }
    return(x)
}

funV <- function(vec){
    return(sapply(vec, ckFun))
}

funD <- function(dat){
    dat$col2 <- sapply(dat$col1, ckFun)
    return(dat)
}
```

ifelse

```
ifeV <- function(vec){
    return(ifelse(vec > 50, vec - 50, vec))
}

ifeD <- function(dat){
    dat$col2 <- ifelse(dat$col1 > 50,
                       dat$col1 - 50,
                       dat$col1)
    return(dat)
}
```

Benchmark Results



More about apply functions

`lapply`, `sapply`, and `mapply` are most common.

lapply

lapply take a list or a vector as an input, applys a function to each element, then returns a list.

Here, we check each element of the list to see if it is numeric or not.

```
myList <- list(1:5,
               LETTERS[1:10],
               c(T, F, T, T))
lapply(myList, is.numeric)
```

```
## [[1]]
## [1] TRUE
##
## [[2]]
## [1] FALSE
##
## [[3]]
## [1] FALSE
```

lapply

Here, we use an anonymous function that checks if the element is numeric and if so sums the element, otherwise it returns the element.

```
lapply(myList, function(x) if(is.numeric(x)) sum(x) else x)
```

```
## [[1]]
## [1] 15
##
## [[2]]
##  [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J"
##
## [[3]]
## [1]  TRUE FALSE  TRUE  TRUE
```

lapply

We can use lapply to subset list elements too.

```
lapply(myList, '[', 1)
```

```
## [[1]]
## [1] 1
##
## [[2]]
## [1] "A"
##
## [[3]]
## [1] TRUE
```

sapply

sapply simplifies the output of lapply. It'll return a vector instead of a list when it can.

```
sapply(myList, '[', 1)
```

```
## [1] "1"    "A"    "TRUE"
```

```
class(sapply(myList, '[', 1))
```

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

sapply

```
sapply(myList, '[', 1:3)
```

```
##      [,1] [,2] [,3]
## [1,] "1"  "A"  "TRUE"
## [2,] "2"  "B"  "FALSE"
## [3,] "3"  "C"  "TRUE"
```

```
class(sapply(myList, '[', 1:3))
```

```
## [1] "matrix"
```

mapply

`mapply` is multi-variable apply. It can take functions with 2+ inputs and map variables to those inputs.

```
letFun <- function() sample(c("A", "B", "C", "7"), 500, replace = T)
slots <- data.frame(fst = letFun(), snd = letFun(), thd = letFun())
print(head(slots))
```

```
##   fst snd thd
## 1   B   A   C
## 2   C   B   7
## 3   B   7   7
## 4   7   A   C
## 5   A   7   A
## 6   C   7   B
```

mapply

```
slotFun <- function(fst, snd, thd){
    if(fst == snd & snd == thd){
        return(paste(fst, snd, thd))
    } else {
        return(NA)
    }
}
runSlots <- mapply(slotFun, slots$fst, slots$snd, slots$thd)
length(which(runSlots == "7 7 7"))
```

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

parallel

Now that you are thinking functionally, imagine this:

A list of elements.

A single funciton.

Assign a portion of those element to each core of a processor and apply the function to those portions.

Re-assemble those elements into a single list once they are complete.