Introduction to Functional programming

David Li

Functions

Functions in R

- A core activity of an R programmer.
- “user” → developer
- When to write a function
  - Encapsulate a sequence of expressions that need to be executed numerous times, perhaps under slightly different conditions.
  - Code must be shared with others or the public
- Create an interface to the code: via a set of parameters.
- This interface provides an abstraction of the code to potential users.
  - Ex: sort()

Your First Function

```
f <- function() {
   ## This is an empty function
}
## Functions have their own class
class(f)
# Execute this function
 f()
```

```
#more fun
f <- function() {
   cat("Hello, world!\n")
}
f()
```

```
#with a parameter
f <- function(num) {
 for(i in seq_len(num)) {
  cat("Hello, world!\n")
  }
}
f(3)
```

```
# with return value
f <- function(num) {
  hello <- "Hello, world!\n"
   for(i in seq_len(num)) {
   cat(hello)
   }
   chars <- nchar(hello) * num
   chars  # logical last expression
returned
}
meaningoflife = f(3)
```

#return the very last expression that is evaluated.

Default value

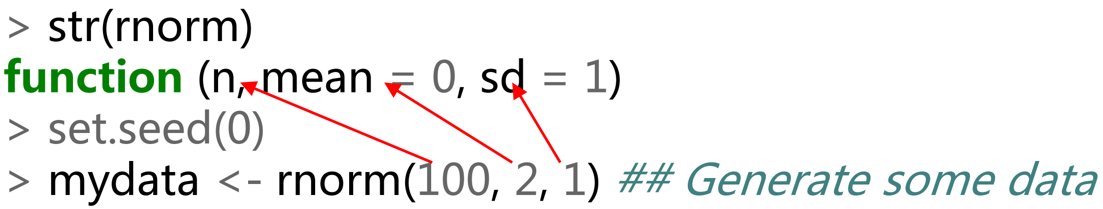
```
f()
f <- function(num = 1) {
  hello <- "Hello, world!\n"
  for(i in seq_len(num)) {
    cat(hello)
  }
  chars <- nchar(hello) * num
  chars
}
f() ## Use default value for 'num  ‘

f(2)
f(num=2) #specified using argument name
```

So far, we have written a function that

- has one *formal argument* named num with a *default value* of 1.
- prints the message “Hello, world!” to the console a number of times indicated by the argument num
- *returns* the number of characters printed to the console

Argument Matching

- R functions arguments can be matched *positionally* or by name.
- Positional matching just means that R assigns the first value to the first argument, the second value to second argument, etc.



```
> str(rnorm)
function (n, mean = 0, sd = 1)
> set.seed(0)
> mydata <- rnorm(100, 2, 1) ## Generate some data
```

100 is assigned to the n argument, 2 is assigned to the mean argument, and 1 is assigned to the sd argument, all by positional matching.

Specifying arguments by name

- Order doesn’ t matter then

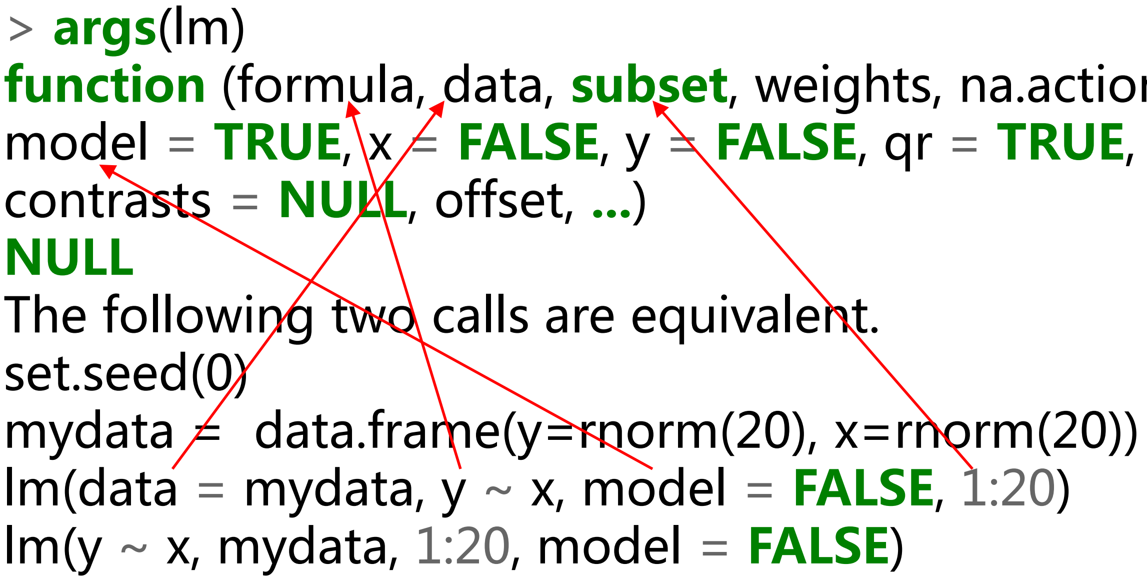
```
> sd(na.rm = FALSE, mydata)
```

Here, the mydata object is assigned to the x argument, because it’s the only argument not yet specified.

- Function arguments can also be *partially* matched

- The order of operations when given an argument
  1. Check for exact match for a named argument
  2. Check for a partial match
  3. Check for a positional match

Example



```
> args(lm)
function (formula, data, subset, weights, na.action, method = "qr",
model = TRUE, x = FALSE, y = FALSE, qr = TRUE, singular.ok = TRUE,
contrasts = NULL, offset, ...)
NULL
```

The following two calls are equivalent.

```
set.seed(0)
mydata =  data.frame(y=rnorm(20), x=rnorm(20))
lm(data = mydata, y ~ x, model = FALSE, 1:20)
lm(y ~ x, mydata, 1:20, model = FALSE)
```

Lazy Evaluation

- Arguments to functions are evaluated *lazily*, so they are evaluated only as needed in the body of the function.

```
> f <- function(a, b) {
   a^2
 }
> f(2)
```

```
> f <- function(a, b) {
  print(a)
  print(b)
}
> f(45)
```

The ... Argument

- A special argument in R
- Indicate a variable number of arguments that are usually passed on to other functions.

```
myplot <- function(x, y, type = "l", ...) {
  plot(x, y, type = type, ...) ## Pass '...' to 'plot' function
}
```

- The ... argument is necessary when the number of arguments passed to the function cannot be known in advance.

```
> args(paste)
function (..., sep = " ", collapse = NULL)
NULL
> args(cat)
function (..., file = "", sep = " ", fill = FALSE, labels = NULL, append = FALSE)
NULL
```

Arguments Coming After the ... Argument

- One catch with ... is that any arguments that appear *after* ... on the argument list must be named explicitly and cannot be partially matched or matched positionally.

```
> args(paste)
function (..., sep = " ", collapse = NULL)
NULL
paste("a", "b", sep = "+")

paste("a", "b", se = "+")
```

Summary

- Functions can be defined using the function() directive and are assigned to R objects just like any other R object
- Functions have can be defined with named arguments; these function arguments can have default values
- Functions arguments can be specified by name or by position in the argument list
- Functions always return the last expression evaluated in the function body
- A variable number of arguments can be specified using the special ... argument in a function definition.

Example: Newton-Rapson method to find a square root of an integer number

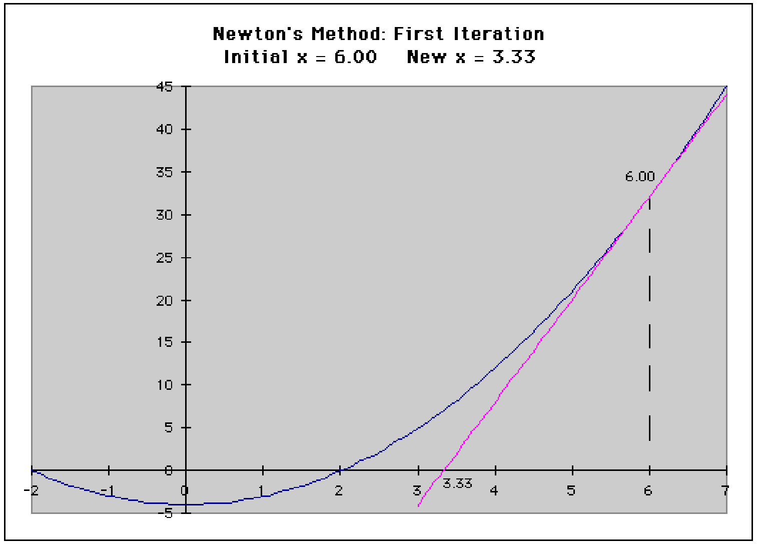
It is an iterative number method. This type of methods create a succecion $(x_0, x_1 \cdots x_n)$.
With some initial conditions given a root of the fucntion f, α

$$f(\alpha) = 0$$

$$\lim_{n->inf} \alpha - x_n = 0$$

More exactly newton-rapson follow the next schema:

$$x_{n+1} = x_n - \frac{f(x)}{f'(x_n)}$$


Newton's Method: First Iteration
Initial x = 6.00 New x = 3.33

For finding a square root of an integer (t) you can do this trick.

$$f(x) = x^2 - t$$

then in our case:

$$x_{n+1} = x_n - \frac{(x_n)^2 - t}{2x_n}$$

where f'(x) = 2x.

Functional Programming

Looping on the Command Line

- apply(): Apply a function over the margins of an array
- lapply(): Loop over a list and evaluate a function on each element
- sapply(): Same as lapply but try to simplify the result
- mapply(): Multivariate version of lapply
- tapply(): Apply a function over subsets of a vector

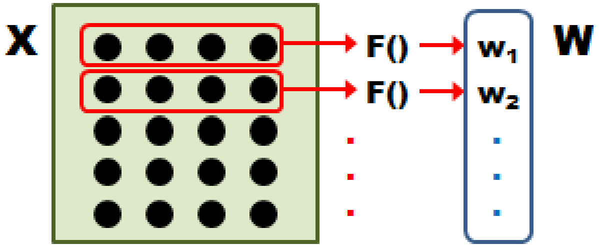
apply()

- Evaluate a function (often an anonymous one) over the margins of an array.
- Most often, apply a function to the rows or columns of a matrix (which is just a 2-dimensional array). Also, applicable to general arrays

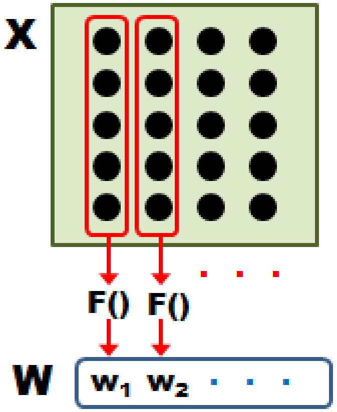
```
> str(apply)
function (X, MARGIN, FUN, ...)
```

- The arguments to apply() are
  - X is an array (matrix is just a 2D array)
  - MARGIN is an integer vector indicating which margins should be “retained” .
  - FUN is a function to be applied
  - ... is for other arguments to be passed to FUN

apply()



```
W <- apply(X, 1, F)
```



```
W <- apply(X, 2, F)
```

apply()

- Examples

```
> set.seed(0)
> x <- matrix(rnorm(200), 20, 10)
> apply(x, 2, mean) ## Take the mean of each column
> apply(x, 1, sum) ## Take the mean of each row
> a <- array(rnorm(2 * 2 * 10), c(2, 2, 10))
> apply(a, c(1, 2), mean)
```

- Shortcuts
  - rowSums = apply(x, 1, sum)
  - rowMeans = apply(x, 1, mean)
  - colSums = apply(x, 2, sum)
  - colMeans = apply(x, 2, mean)

lapply()

- The lapply() function does the following simple series of operations:

1. it loops over a list, iterating over each element in that list
2. it applies a *function* to each element of the list (a function that you specify)
3. and returns a list (the l is for "list").

- This function takes three arguments
  - (1) a list X, If X is not a list, it will be coerced to a list using as.list().
  - (2) a function (or the name of a function) FUN;
  - (3) other arguments via its ... argument.

lapply()

- the actual looping is done internally in C code for efficiency reasons.

```
> lapply
function (X, FUN, ...)
{
FUN <- match.fun(FUN)
if (!is.vector(X) || is.object(X))
X <- as.list(X)
.Internal(lapply(X, FUN))
}
<bytecode: 0x7fcc8388f758>
<environment:
namespace:base>
```

lapply()

- Example 1

```
> x <- list(a = 1:5, b = rnorm(10))
> lapply(x, mean)
```

```
> x <- list(a = 1:4, b = rnorm(10), c = rnorm(20, 1), d = rnorm(100, 5))
> lapply(x, mean)
```

- Example 2

```
> x <- 1:4
> lapply(x, runif)
```

When you pass a function to lapply(), lapply() takes elements of the list and passes them as the *first argument* of the function you are applying.

lapply()

- the … Argument

```
> x <- 1:4
> lapply(x, runif, min = 0, max = 10)
```

- *anonymous* functions.

```
> x <- list(a = matrix(1:4, 2, 2), b = matrix(1:6, 3, 2))

> lapply(x, function(elt) { elt[,1] })
```

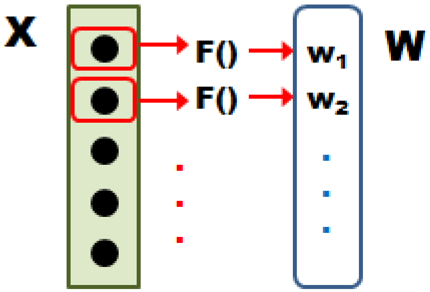
V.S.

```
> f <- function(elt) {
    elt[, 1]
  }
> lapply(x, f)
```

sapply()

- The sapply() function behaves similarly to lapply(); the only real difference is in the return value.

W <- lapply(X, F)
W <- sapply(X, F)



- sapply() will try to simplify the result of lapply() if possible. Essentially, sapply() calls lapply() on its input and then applies the following algorithm
  - If the result is a list where every element is length 1, then a vector is returned
  - If the result is a list where every element is a vector of the same length (> 1), a matrix is returned.
  - If it can' t figure things out, a list is returned

```
> x <- list(a = 1:4, b = rnorm(10), c = rnorm(20, 1), d = rnorm(100, 5))
> lapply(x, mean)

> sapply(x, mean)
```

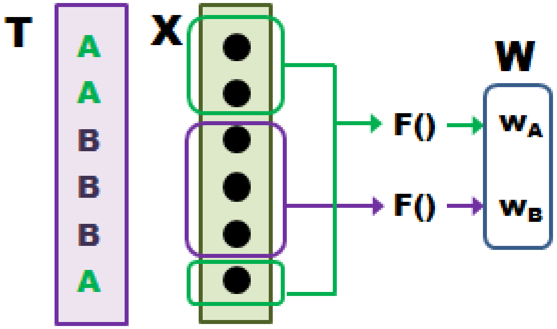
split()

- The combination of split() and a function like lapply() or sapply() is a common paradigm in R.

```
> library(datasets)
> head(airquality)

> s <- split(airquality, airquality$Month)
> str(s)

> lapply(s, function(x) {
  colMeans(x[, c("Ozone", "Solar.R", "Wind")])
  })

> sapply(s, function(x) {
   colMeans(x[, c("Ozone", "Solar.R", "Wind")])
 })

> sapply(s, function(x) {
  colMeans(x[, c("Ozone", "Solar.R", "Wind")],na.rm = TRUE)
 })
```

tapply

- tapply() is used to apply a function over subsets of a vector. It can be thought of as a combination of split() and sapply() for vectors only.



W <- tapply(X, T, F)

```
> str(tapply)
function (X, INDEX, FUN = NULL, ..., simplify = TRUE)
```

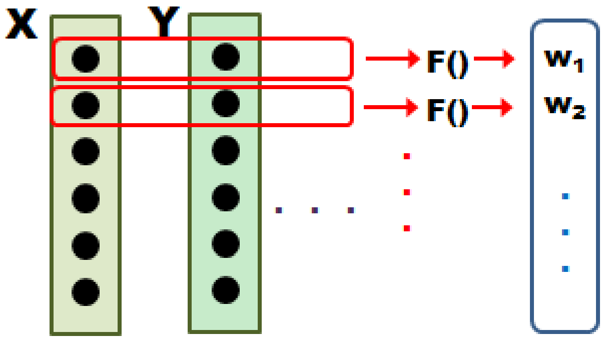
- The arguments to tapply() are as follows:
  - X is a vector
  - INDEX is a factor or a list of factors (or else they are coerced to factors)
  - FUN is a function to be applied
  - ... contains other arguments to be passed FUN
  - simplify, should we simplify the result?

tapply

```
> ## Simulate some data
> x <- c(rnorm(10), runif(10), rnorm(10, 1))
> ## Define some groups with a factor variable
> f <- gl(3, 10)
> f
[1] 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 3 3 3 3 3 3 3 3 3 3
Levels: 1 2 3
> tapply(x, f, mean)


> tapply(x, f, mean, simplify = FALSE)
> tapply(x, f, mean, simp = FALSE) #will it return the same? why
# when returning >1 value. tapply() will not simplify the result and
# will return a list.
> tapply(x, f, range)
```

mapply()

- A multivariate apply of sorts which applies a function in parallel over a set of arguments.
- Recall that lapply() and friends only iterate over a single R object. What if you want to iterate over multiple R objects in parallel? This is what mapply() is for.



W <- mapply(F, X, Y, ...)

```
> str(mapply)
function (FUN, ..., MoreArgs = NULL, SIMPLIFY = TRUE, USE.NAMES =
TRUE)
```

The arguments to mapply() are

- FUN is a function to apply
- ... contains R objects to apply over
- MoreArgs is a list of other arguments to FUN.
- SIMPLIFY indicates whether the result should be simplified

mapply()

```
list(rep(1, 4), rep(2, 3), rep(3, 2), rep(4, 1))

> mapply(rep, 1:4, 4:1)

> noise <- function(n, mean, sd) {
 rnorm(n, mean, sd)
}
> ## Simulate 5 random numbers
> noise(5, 1, 2)
[1] -0.5196913 3.2979182 -0.6849525 1.7828267 2.7827545
>
> ## This only simulates 1 set of numbers, not 5
> noise(1:5, 1:5, 2)
[1] -1.670517 2.796247 2.776826 5.351488 3.422804

> mapply(noise, 1:5, 1:5, 2)     ==   list(noise(1, 1, 2), noise(2, 2, 2),
                                       noise(3, 3, 2), noise(4, 4, 2),
                                       noise(5, 5, 2))
```

Vectorizing a Function

- The mapply() function can be used to automatically “vectorize” a function: take a function that typically only takes single arguments and create a new function that can take vector arguments.

```
> sumsq <- function(mu, sigma, x) {
   sum(((x - mu) / sigma)^2)
  }
> x <- rnorm(100) ## Generate some data
> sumsq(1:10, 1:10, x) ## This is not what we want
[1] 110.2594
```

However, we can do what we want to do by using mapply().

```
> mapply(sumsq, 1:10, 1:10, MoreArgs = list(x = x))
```

Vectorize()

- It can automatically create a vectorized version of your function.
- Example: create a vsumsq() function that is fully vectorized as follows.

```
> vsumsq <- Vectorize(sumsq, c("mu", "sigma"))
> vsumsq(1:10, 1:10, x)
[1] 196.2289 121.4765 108.3981 104.0788 102.1975 101.2393 100.6998
[8] 100.3745 100.1685 100.0332
```

Summary

- The loop functions in R are very powerful because they allow you to conduct a series of operations on data using a compact form
- The operation of a loop function involves iterating over an R object (e.g. a list or vector or matrix), applying a function to each element of the object, and the collating the results and returning the collated results.
- Loop functions make heavy use of anonymous functions, which exist for the life of the loop function but are not stored anywhere
- The split() function can be used to divide an R object in to subsets determined by another variable which can subsequently be looped over using loop functions.

Computing Lab Ex

- Further Readings
  - http://adv-r.had.co.nz/Functionals.html
  - https://towardsdatascience.com/functional-programming-in-r-with-purrr-469e597d0229
- Computing Lab Ex
  - Try to run the sample code in notebook
  - Speed up the loop operation in R
    - http://stackoverflow.com/questions/2908822/speed-up-the-loop-operation-in-r
  - Is R's apply family more than syntactic sugar?
    - http://stackoverflow.com/questions/2275896/is-rs-apply-family-more-than-syntactic-sugar