Programming for Data Analytics

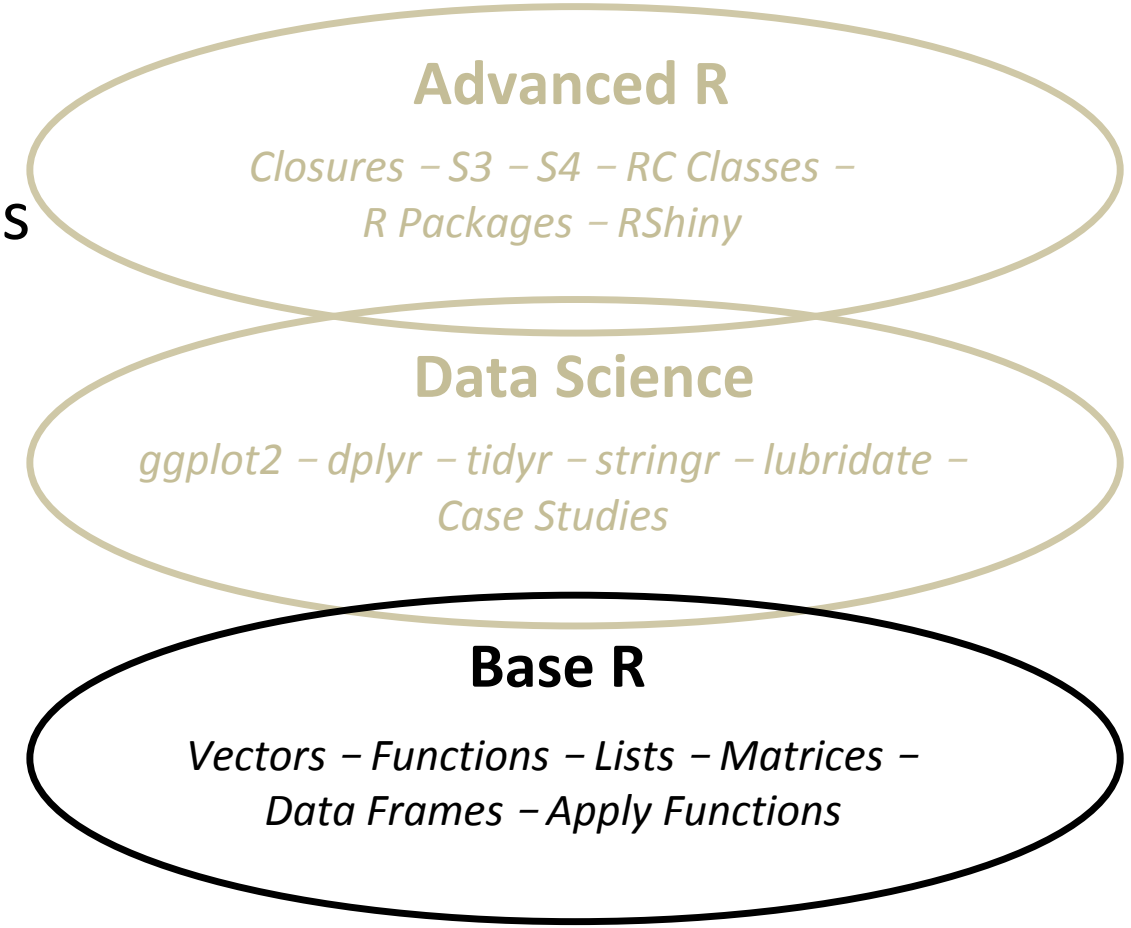
3. Matrices and Functionals

Dr. Jim Duggan,
School of Engineering & Informatics
National University of Ireland Galway.

https://twitter.com/_jimduggan

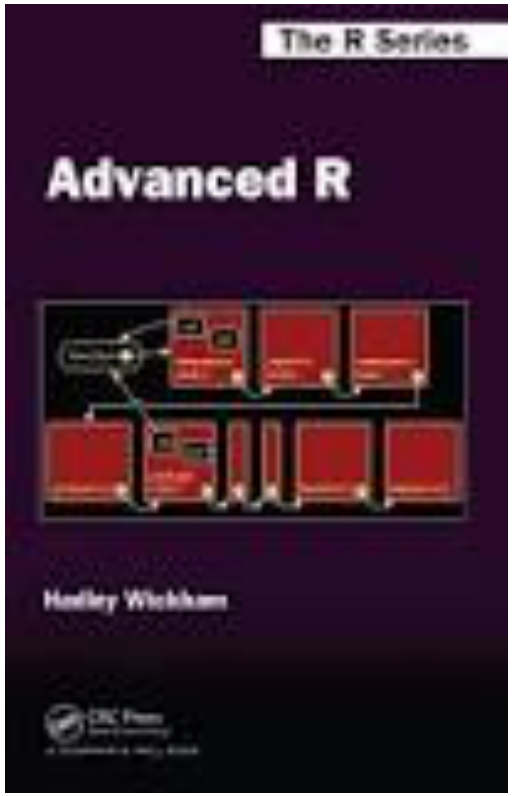
Lecture 02 - Overview

1. More on functions
2. Matrices
3. Functionals



(1) More on Functions

- …
- do.call()
- Replacement functions



… argument

- There is a special argument called **…**
- This argument will match any arguments not otherwise matched, and can be easily passed on to other functions
- This is useful if you want to collect arguments to call another function, but you don't want to prespecify their possible names

```
mysum <- function(v, ...){
  sum(v,...)
}

> x <- c(10, 20, 30 ,NA)
>
> x
[1] 10 20 30 NA
>
> mysum(x)
[1] NA
>
> mysum(x, na.rm=T)
[1] 60
```

do.call()

- Calling a function, given a list of arguments
- With a list of arguments, we can send these to a function

```
> args <- list(c(1:10,NA), na.rm = TRUE)
>
> args
[[1]]
 [1]  1  2  3  4  5  6  7  8  9 10 NA

$na.rm
[1] TRUE

>
> ans <- do.call(sum,args)
>
> ans
[1] 55
```

Replacement Functions

- Replacement functions act like they modify their arguments in place, and have the special name `` `xxx<-` ``
- They typically have two arguments (x and value), although they can have more, and they must return the modified object
- The new value must be passed as a parameter named **value**

```
1 `second<-` <- function(x, value) {
2   x[2] <- value
3   x
4 }
```

```
> x <- 1:10
>
> x
 [1]  1  2  3  4  5  6  7  8  9 10
>
> second(x) <- 78
>
> x
 [1]  1 78  3  4  5  6  7  8  9 10
```

Challenge 3.1

- Create a replacement function that performs the reverse of the names() function.

```
> x<-1:5
>
> x
[1] 1 2 3 4 5
>
> n <- letters[x]
>
> n
[1] "a" "b" "c" "d" "e"
>
> rev_names(x)<-n
>
> x
e d c b a
1 2 3 4 5
```

(2) Matrices

	Homogenous	Heterogenous
1d	Atomic Vector	List
2d	**Matrix**	Data Frame
nd	Array	

- A matrix can be initialized from a vector, where the numbers of rows and columns are specified.
- R stores matrices by column-major order, and by default matrices are filled in this manner.

Declaring a matrix

```
>
> a <- matrix(1:6, ncol=3, nrow=2)
>
> a
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
>
> dim(a)
[1] 2 3
```

*Note: **dim** is also a replacement function and can be used to set the matrix dimensions. Try and change the matrix a to a 3 x 2 matrix*

Adding rows and columns

```
>
> cbind(a,c(7,8))
     [,1] [,2] [,3] [,4]
[1,]    1    3    5    7
[2,]    2    4    6    8
>
>
> rbind(a,c(7,8,9))
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
[3,]    7    8    9
```

Naming rows and columns

```
> rownames(a) <- c("A","B")
>
> a
  [,1] [,2] [,3]
A    1    3    5
B    2    4    6
>
> colnames(a) <- c("a","b","c")
>
> a
  a b c
A 1 3 5
B 2 4 6
```

Subsetting Matrices

- The most common way of subsetting 2d matrix is a simple generalisation of 1d subsetting
- Supply a 1d index for each dimension, separated by a comma
- Blank subsetting is useful, as it lets you keep all rows or all columns

Using row index…

```
> b <- matrix(1:9, nrow=3)
>
> colnames(b) <- c("A","B","C")
>
> b
     A B C
[1,] 1 4 7
[2,] 2 5 8
[3,] 3 6 9
>
> b[1:2,]
     A B C
[1,] 1 4 7
[2,] 2 5 8
```

```
> b[c(T,F),]
     A B C
[1,] 1 4 7
[2,] 3 6 9
>
>
> b[-3,]
     A B C
[1,] 1 4 7
[2,] 2 5 8
```

Using column index…

```
> b
     A B C
[1,] 1 4 7
[2,] 2 5 8
[3,] 3 6 9
>
> b[,1:2]
     A B
[1,] 1 4
[2,] 2 5
[3,] 3 6
```

```
> b[,c(T,F)]
     A C
[1,] 1 7
[2,] 2 8
[3,] 3 9
>
> b[,c("A","C")]
     A C
[1,] 1 7
[2,] 2 8
[3,] 3 9
```

Sample Matrix Operations

Operator or Function	Description
A * B	Element-wise multiplication
A / B	Element-wise division
A %*% B	Matrix multiplication
t(A)	Transpose of A
e<-eigen(A)	List of eigenvalues and eigenvectors for matrix A

Challenge 3.2

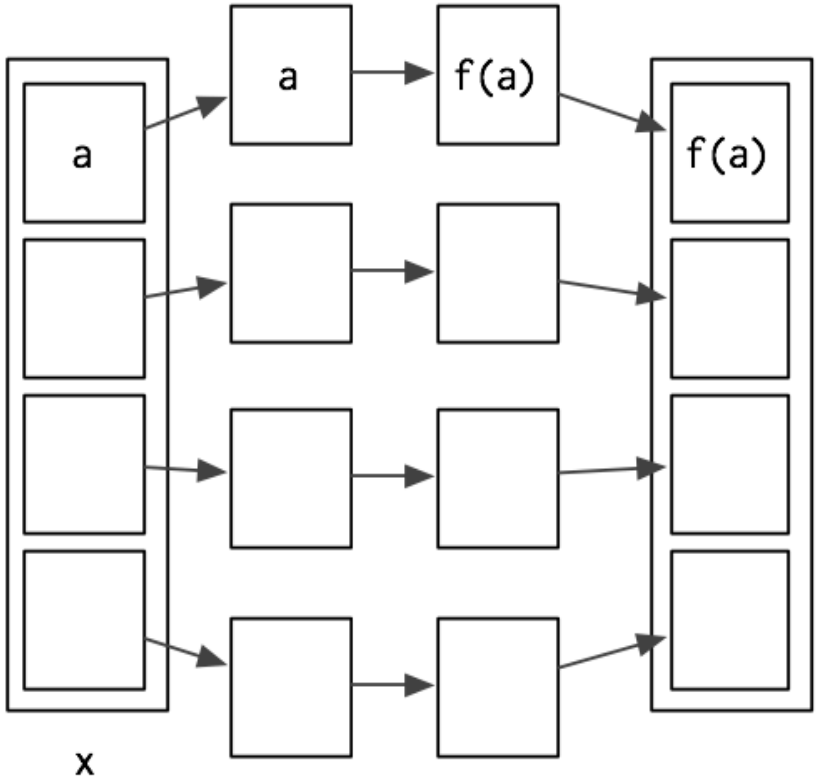
- Create a 10x10 matrix to represent connections between people on social media (random seed=100)
- Label the people [A..J], with named rows and columns.
- Randomly populate the matrix with 1s and 0s. The number 1 means someone follows/is followed by another person. Ensure that all diagonals are 0 (you should use an appropriate R matrix operation for this).
- Each row contains information on the people a person follows. For example, {A} *follows* {C,D,G,J}
- Each column contains information on who follows a person. For example {A} *is followed by* {D,E,F,H}

(3) Functionals

- **A functional** is a function that takes a function as an input and returns a vector as output
- Commonly used as an alternative for loops
- Loops are not very expressive, they do not convey a higher-level goal

my_lapply(x,f)

- Common pattern:
  - Create a container for output
  - Apply f() to each component of the list
  - Fill the container with the results



Looping Patterns

There are three basic ways to loop over a vector:
loop over the elements: for (x in xs)

Method	Syntax	Comments
loop over the elements	for (x in xs)	not a good choice for a for loop because it leads to inefficient ways of saving output
loop over the numeric indices	for (i in seq_along(xs))	Allows you to to create the space you'll need for the output and then fill it in.
loop over the names	for (nm in names(xs))	

```
3  my_lapply <- function(x, f, ...) {
4    out <- vector("list", length(x))
5    for (i in seq_along(x)) {
6      out[[i]] <- f(x[[i]], ...)
7    }
8    out
9  }
```

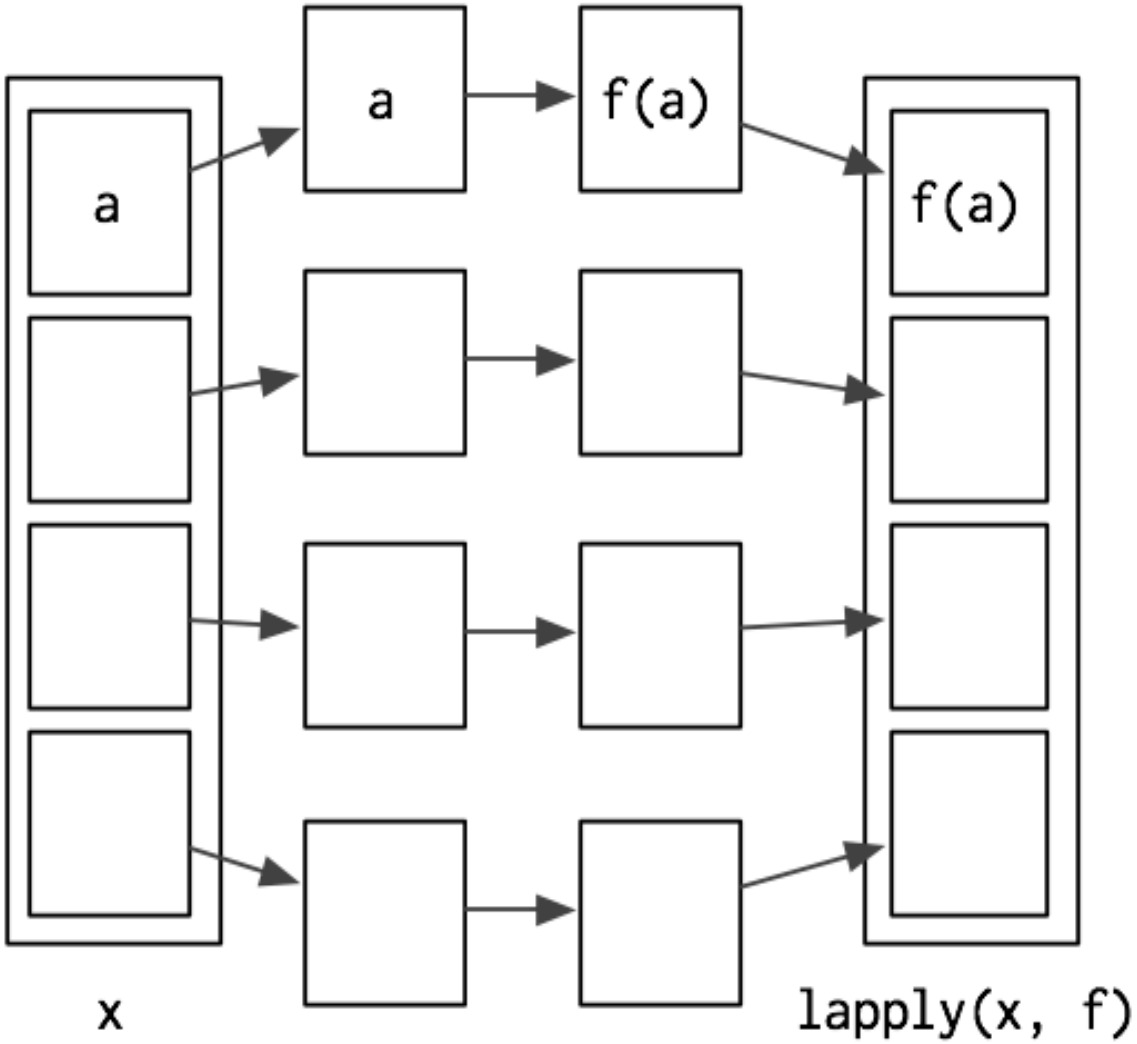
```
> l <- list(1:2, 3:4, c(4:10,NA))
> str(l)
List of 3
 $ : int [1:2] 1 2
 $ : int [1:2] 3 4
 $ : int [1:8] 4 5 6 7 8 9 10 NA
```

```
> ans<-my_lapply(l,sum,na.rm=T)
> str(ans)
List of 3
 $ : int 3
 $ : int 7
 $ : int 49
```

R functional - lapply(x,f)

- Another use of user-defined functions in R is as parameters to the *apply* family of functions.
- The general form of the **lapply(x,f,fargs)** function is as follows:
  - **x** is the target vector or list
  - **f is the function to be called and applied to each element**
  - **fargs** are the optional set of arguments that can be applied to the function f.
  - **lapply()** returns a list

Overall structure



Challenge 3.3

- Debug the following code to see what is passed in each time (hint use the browser() call)

```
xs <- list(el1=1:5, el2=c(T,F),el3=100:110)

lapply(xs, function(x) {x})

lapply(seq_along(xs), function(i) {xs[[i]]})

lapply(names(xs), function(nm) {xs[[nm]]})
```

R functional - sapply(x,f)

- Another use of user-defined functions in R is as parameters to the *apply* family of functions.
- The general form of the **sapply(x,f,fargs)** function is as follows:
  - **x** is the target vector or list
  - **f is the function to be called and applied to each element**
  - **fargs** are the optional set of arguments that can be applied to the function f.
  - **sapply()** returns a vector

Challenge 3.4

- Use the interval [-10,10] for x, and the **seq()** function to generate the values in steps of 0.1
- Generate the response f(x) by calling sapply()
- Plot the response using qplot()

$$f(x) = ax^2 + bx + c$$

R functional - apply()

- The **apply()** function can be used to process rows and columns for a matrix, and the general form of this function (Matloff 2009) is **apply(m, dimcode, f, fargs)**, where:
  - **m** is the target matrix
  - **dimcode** identifies whether it's a row or column target. The number 1 applies to rows, whereas 2 applies to columns
  - **f** is the function to be called
  - **fargs** are the optional set of arguments that can be applied to the function **f**.

Examples…

```
> b
     A B C
[1,] 1 4 7
[2,] 2 5 8
[3,] 3 6 9
>
> apply(b,1,sum)
[1] 12 15 18
```

```
> b
     A B C
[1,] 1 4 7
[2,] 2 5 8
[3,] 3 6 9
>
> apply(b,2,sum)
 A  B  C
 6 15 24
```

Challenge 3.5

- Create a matrix of grades for subjects
- Each column is a different subject, with grades drawn from a random normal distribution
- Each row is a students result
- Use apply to calculate the average mark for each student, and add this result as a new column to the matrix.

```
1 set.seed(10)
2 N=10
3 cs1 <- rnorm(N,72,10)
4 cs2 <- rnorm(N,65,7)
5 cs3 <- rnorm(N,80,9)
6 cs4 <- rnorm(N,55,7)
7 cs5 <- rnorm(N,61,5)
```

```
          cs1      cs2      cs3      cs4      cs5       av
 [1,] 72.18746 72.71246 74.63320 42.02382 66.43276 65.59794
 [2,] 70.15747 70.29047 60.33242 54.45438 57.18728 62.48440
 [3,] 58.28669 63.33237 73.92621 61.77996 56.85669 62.83638
 [4,] 66.00832 71.91211 60.92845 56.29448 65.17237 64.06315
 [5,] 74.94545 70.18973 68.61322 45.34039 56.16174 63.05011
 [6,] 75.89794 65.62543 76.63705 44.95140 60.85592 64.79355
 [7,] 59.91924 58.31539 73.81200 57.53461 62.16263 62.34877
 [8,] 68.36324 63.63395 72.15057 42.68639 59.49396 61.26562
 [9,] 55.73327 71.47865 79.08415 52.72819 57.61193 63.32724
[10,] 69.43522 68.38085 77.71598 50.43906 64.27614 66.04945
```