Introduction to Functional Programming in R

M.H. Ferdosi

Animal Genetics and Breeding Unit
University of New England, Armidale

Programming paradigm

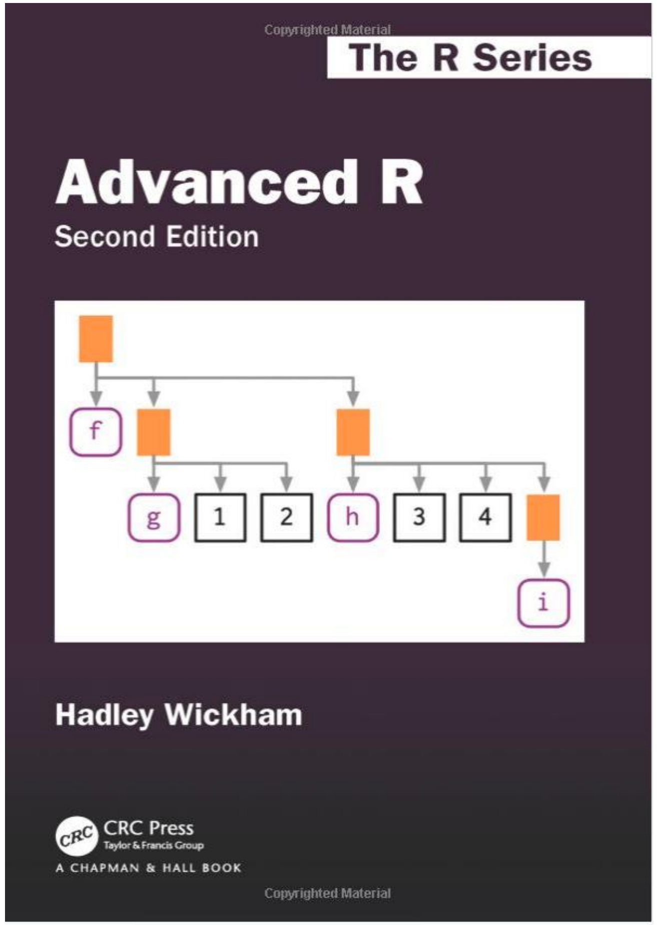
- Classifying programming languages by their features (style)
- Major types of programming paradigms:
- Generic → templates (variable types specified later)
- Imperative
  - Object oriented → object, class, method, member
  - Procedural → based on procedure, routines, subroutines or functions
- Declarative
  - Functional

Functional Programming

- “In computer science, functional programming is a programming paradigm—a style of building the structure and elements of computer programs—that treats computation as the evaluation of mathematical functions and avoids changing-state and mutable data.” wikipedia

Functional programming in R

- Advanced R, Second Edition
- Hadley Wickham
    - The author of ggplot2, readr, dplyr, reshape2 and …
- https://adv-r.hadley.nz/fp.html



Functional Programming Concepts

- Pure Functions
- Recursion
- First class and higher order functions
- …

Pure Functions

- Same input → Same output

- No side effect → change values on disk or global variable

- …
  - How can we set a global variable in R?

Examples of pure and impure functions in R

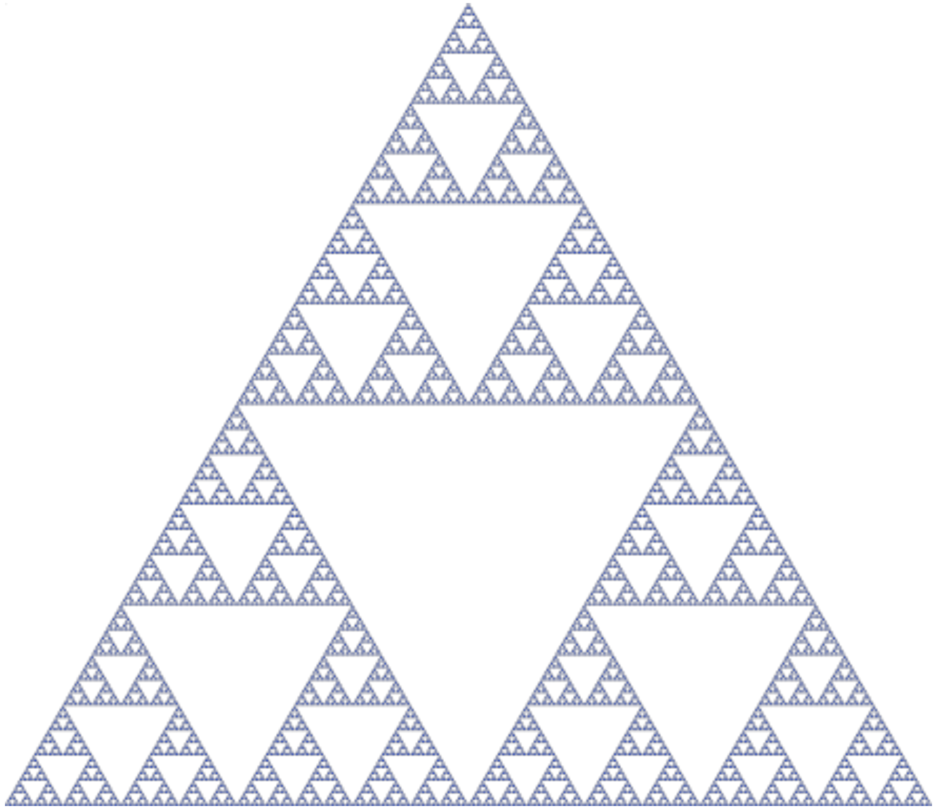
- Pure function:
  - average = function(x) {mean(x)}
- Impure function
  - A = 3
  - Add = function(x) {A <<- A + x; return(A)}
  - Add(2)
  - Add(2)

Results

```
> Add = function(x) {A <<- A + x; return(A)}
> Add(2)
[1] 5
> Add(2)
[1] 7
```

Recursion

- A function call itself recursively

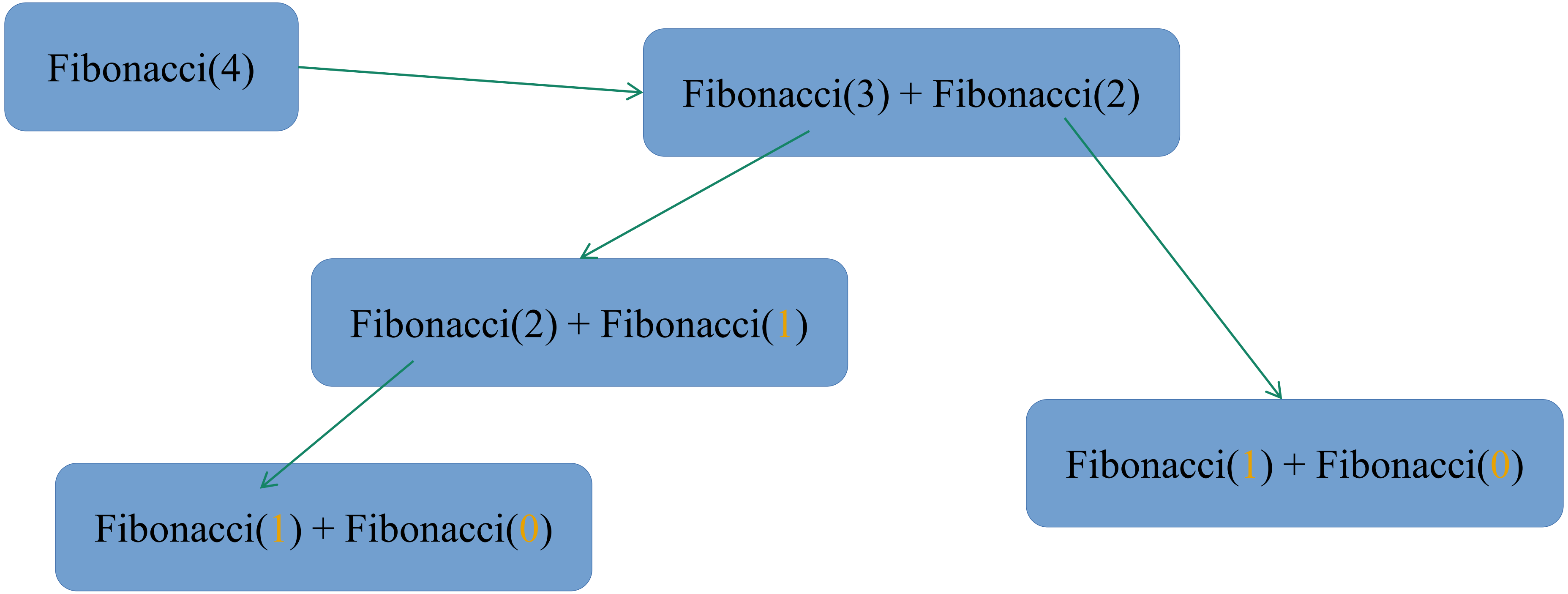


Ref: Wikipedia

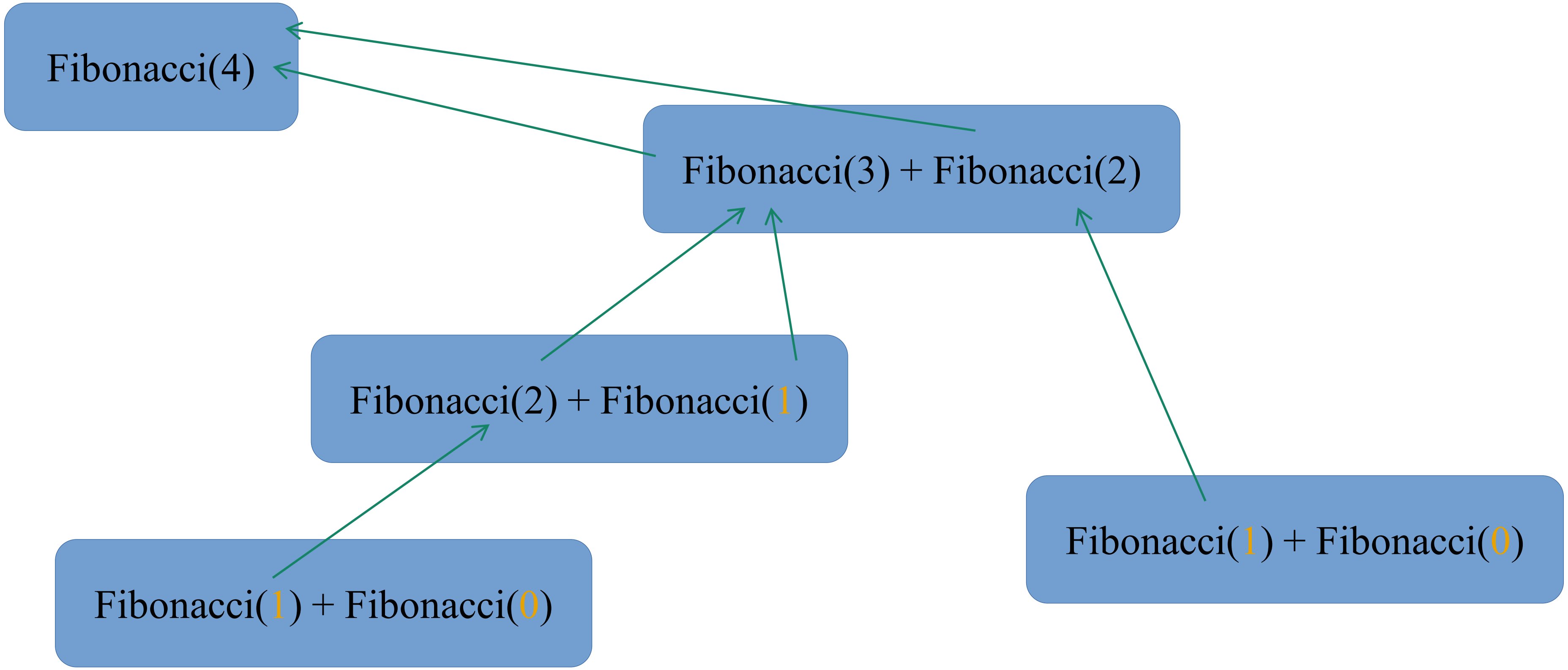
Recursion Example

```
• Fibonacci = function(x)
{

      if(x == 0)

        return (0)

       if(x == 1)

        return (1)

       return(Fibonacci(x - 1) + Fibonacci(x - 2)) # Fibonacci being called again by itself.
}
• Fibonacci(3)
• Fibonacci(5)
```

Recursion Flowchart (First Part)


Fibonacci(4)
Fibonacci(3) + Fibonacci(2)
Fibonacci(2) + Fibonacci(1)
Fibonacci(1) + Fibonacci(0)
Fibonacci(1) + Fibonacci(0)

Recursion Flowchart (return)



Fibonacci(4)

Fibonacci(3) + Fibonacci(2)

Fibonacci(2) + Fibonacci(1)

Fibonacci(1) + Fibonacci(0)

Fibonacci(1) + Fibonacci(0)

Do you have an example in your field?

First class and higher order functions

- Functions can be used as argument of another function
  - For example

```
sapply(1:10, sin)
```

Functional style

- Regular Function
- Functionals
- Function factories
- Function operators

Regular Function

- Data in → Data out
- Example

```
SampleFun = function(x)
{
    stopifnot(!is.character(x))
    return(x^2)
}
```

- SampleFun(5)

Functionals

- Using function as argument of another function
  - My10Number <- function(fun) fun(1:10)
    - class(My10Number)
    - typeof(My10Number)
  - My10Number(mean)
  - My10Number(sd)
  - My10Number(log)

Results

- (My10Number <- function(fun) fun(1:10))
    - class(My10Number)
        - "function"
    - typeof(My10Number)
        - "closure"
  - My10Number(mean)
    - [1] 5.5
  - My10Number(sd)
    - [1] 3.02765
  - My10Number(log)

```
[1] 0.0000000 0.6931472 1.0986123 1.3862944 1.6094379 1.7917595 1.9459101
[8] 2.0794415 2.1972246 2.3025851
```

Map function

- “ ‘Map’ is a simple wrapper to ‘mapply’ which does not attempt to simplify the result” R help
  - (A = Map(log, 1:10))
  - class(A)
  - (B = mapply(log, 1:10))
  - class(B)

Results (A)

```
> (A = Map(log, 1:10))
[[1]]
[1] 0
[[2]]
[1] 0.6931472
> class(A)
[1] "list"
```

Results (B)

```
> (B = mapply(log, 1:10))
 [1] 0.0000000 0.6931472 1.0986123 1.3862944 1.6094379
1.7917595 1.9459101
 [8] 2.0794415 2.1972246 2.3025851
> class(B)
[1] "numeric"
```

Reduce function

- Reduce(f = "+", x = 1:10, accumulate = TRUE)
- Reduce(f = "+", x = 1:10, accumulate = FALSE)
- Reduce(f = "-", x = 1:10, accumulate = TRUE)
- Reduce(f = paste, x = 1:10, accumulate = TRUE)

Results

```
> Reduce(f = "+", x = 1:10, accumulate = TRUE)
 [1]  1  3  6 10 15 21 28 36 45 55
> Reduce(f = "+", x = 1:10, accumulate = FALSE)
[1] 55
> Reduce(f = "-", x = 1:10, accumulate = TRUE)
 [1]   1  -1  -4  -8 -13 -19 -26 -34 -43 -53
> Reduce(f = paste, x = 1:10, accumulate = TRUE)
 [1] "1"                    "1 2"                  "1 2 3"
 [4] "1 2 3 4"              "1 2 3 4 5"            "1 2 3 4 5 6"
 [7] "1 2 3 4 5 6 7"        "1 2 3 4 5 6 7 8"      "1 2 3 4 5 6 7 8 9"
[10] "1 2 3 4 5 6 7 8 9 10"
```

Reduce function (common elements in vectors)

- Find the elements that are common in all these vectors:
  - set.seed(2)
  - L = mapply(sample, rep(list(1:10), 10), 20, replace = TRUE, SIMPLIFY = FALSE)
  - Reduce(f = intersect, L)

Results

```
> L
[[1]]
 [1] 5 6 6 8 1 1 9 2 1 3 6 2 3 7 8 7 1 6 9 4
[[2]]
 [1] 6 9 8 6 3 9 7 8 6 2 7 2 3 4 3 1 7 9 1 2
> Reduce(f = intersect, L)
[1] 6 8 7 4
```

Filter function

- Simulate a data frame:
  - Data = data.frame(A = 1:3, B = letters[1:3], C = runif(3))
  - lapply(Data, class)
  - apply(Data, 2, class)

Filter function

```
> (Data = data.frame(A = 1:3, B = letters[1:3], C =
runif(3)))
  A B         C
1 1 a 0.1883563
2 2 b 0.8731385
3 3 c 0.9811036
```

```
> lapply(Data, class)
$A
[1] "integer"
$B
[1] "factor"
$C
[1] "numeric"
> apply(Data, 2, class)
          A           B           C
"character" "character" "character"
```

Filter function

```
1) Filter(function(x) !is.numeric(x), Data)
2) Filter(function(x) is.numeric(x), Data)
3) Filter(function(x) is.character(x), Data)
4) Filter(function(x) is.factor(x), Data)
5) Filter(function(x) sum(as.numeric(x)) > 10, Data)
6) Filter(function(x) sum(as.numeric(x)) > 10, 1:20)
7) (Ind = Position(function(x) sum(as.numeric(x)) > 10, 2:20))
8) Find(function(x) sum(as.numeric(x)) > 10, 2:20); T = 2:20; T[Ind]
```

Apply family of functions

- apply
- sapply
- rapply
- vapply
- lapply
- mapply
- tapply

Apply function

- Apply a function on the rows or columns of matrix or data.frame (array)
- A = matrix(1:10, nrow = 5)
  - apply(A, 1, sum)
  - apply(A, 2, sum)

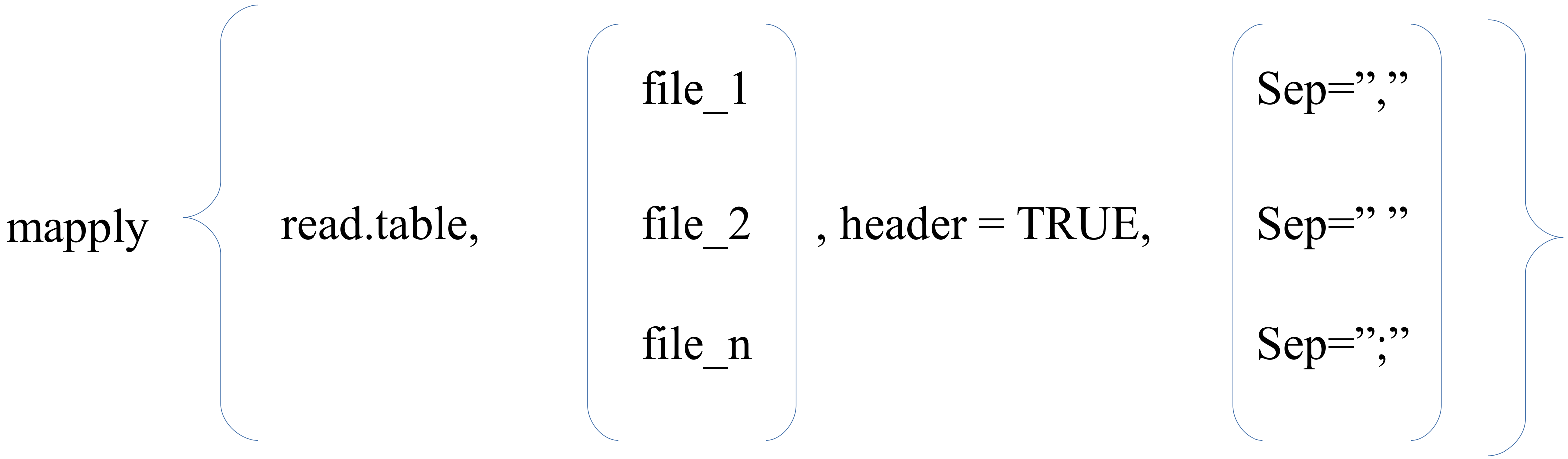
sapply, lapply and vapply functions

- Apply a function on each element
- sapply(1:10, log); class(sapply(1:10, log))
- lapply(1:10, log); class(lapply(1:10, log))
- vapply(1:10, log, FUN.VALUE = double(1))
- vapply(1:10, log, FUN.VALUE = numeric(1))
- vapply(1:10, log, FUN.VALUE = integer(1))
- vapply(1:10, log, FUN.VALUE = character(1))

sapply, lapply and vapply functions

- FunA = function(){matrix(sample(1:10, 10), nrow = 5)}
- set.seed(1); B = list(FunA(), FunA(), FunA())
- Res_1 = lapply(B, rowSums); class(Res_1)
- Res_2 = sapply(B, rowSums); class(Res_2)
- Res_3 = do.call(rbind, Res_1)
- Res_4 = do.call(cbind, Res_1)

mapply



mapply

read.table,

file_1

file_2

file_n

, header = TRUE,

Sep=”,”

Sep=” ”

Sep=”;”

- mapply(sum, Res_1)
- Check the results
  - lapply(Res_1, sum)

tapply function

- Apply a function on each group of data
- set.seed(1); Data = data.frame(A = 1:10, B = sample(1:3,10,replace=TRUE), C = runif(10))
- tapply(Data$A, Data$B, sum)

Result

```
> tapply(Data$A, Data$B, sum)
1  2  3
9 21 25
```

Function factories

- A function that take data as input and returns a function/s

Function factories - Example

-
  ```
  sumsum <- function(x)
  {
      newFun <- function(y)
      {
          x + y
      }
      return(newFun)
  }
  ```
- sumsum(5)(3)
- Afun = sumsum(2)
- Afun(5)

Results

```
>sumsum(5)(3)
[1] 8
>Afun = sumsum(2)
> Afun(5)
[1] 7
```

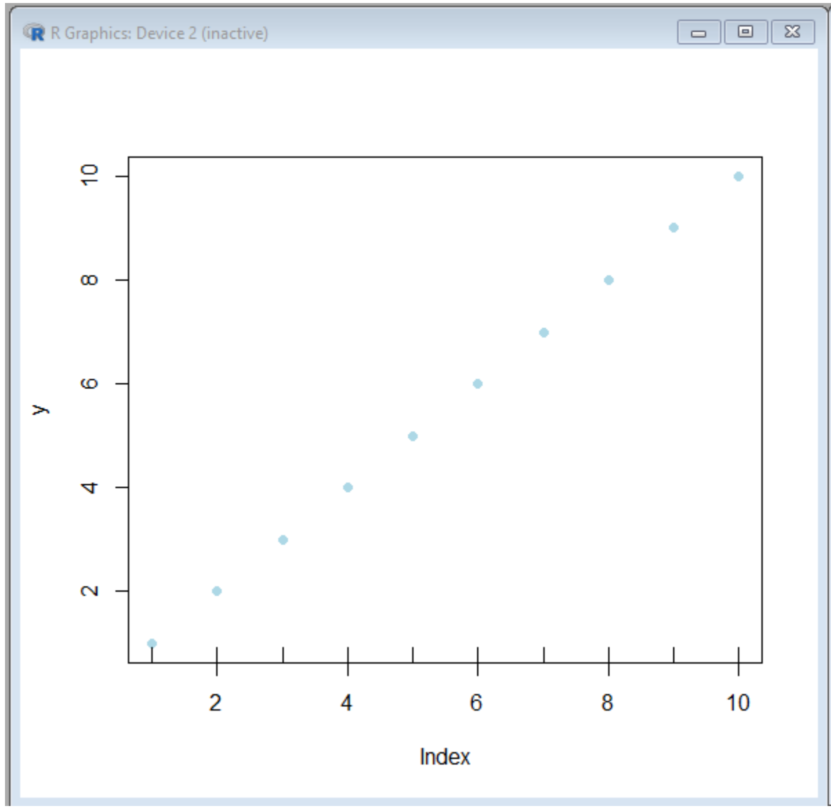
Function operators

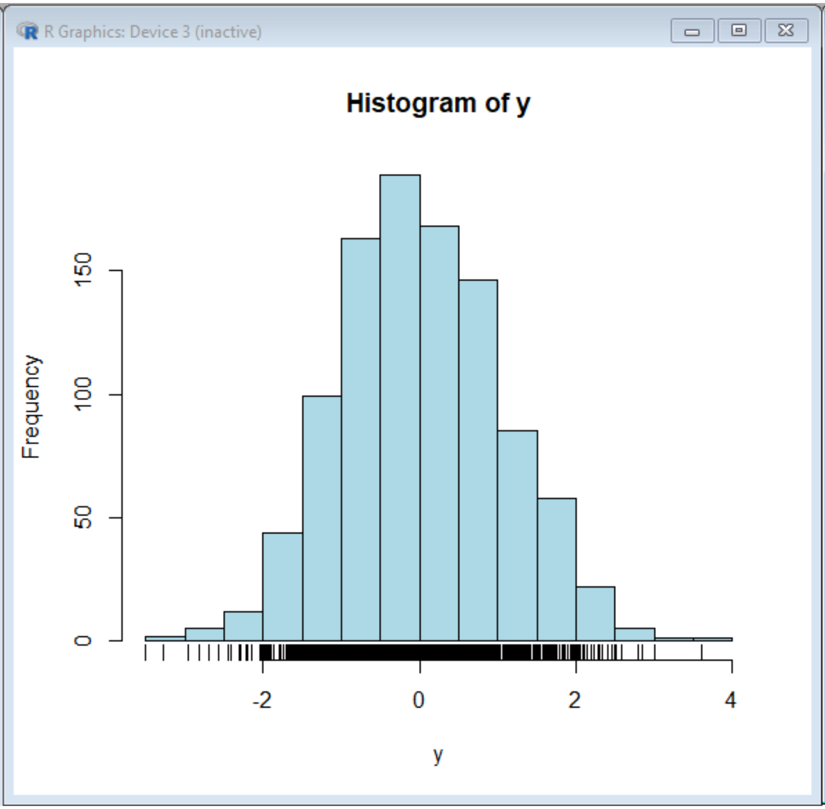
- A function that take function/s as argument/s and return a function as output.

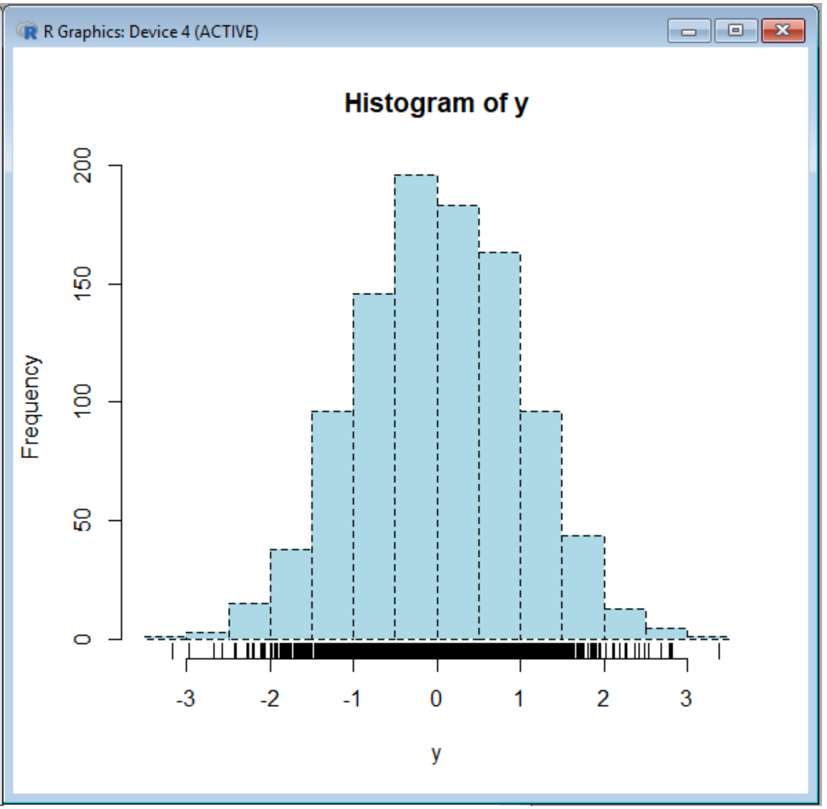
Function operators - Example

```
plotRev <- function(myFUN)
{
        function(y,...)
        {
                res <- myFUN(y, ... , col = "lightblue", pch = 16)
                rug(y)
                return(res)
        }
}
plotRev(plot)(1:10)
plotRev(hist)(rnorm(1000))
plotRev(hist)(rnorm(1000), lty=2)
```

Function operators - Results







Thanks for your attention!