

Un poco sobre funciones. . .

Jorge Loría

Sept 11, 2017

Recapitulación clase pasada

Vimos que existen 5 tipos básicos de estructuras:

- ▶ Vectores

Recapitulación clase pasada

Vimos que existen 5 tipos básicos de estructuras:

- ▶ Vectores
- ▶ Listas

Resumen de la clase pasada

Vimos que existen 5 tipos básicos de estructuras:

- ▶ Vectores
- ▶ Listas
- ▶ Matrices

Recapitulación clase pasada

Vimos que existen 5 tipos básicos de estructuras:

- ▶ Vectores
- ▶ Listas
- ▶ Matrices
- ▶ DataFrames *

Recapitulación clase pasada

Vimos que existen 5 tipos básicos de estructuras:

- ▶ Vectores
- ▶ Listas
- ▶ Matrices
- ▶ DataFrames *
- ▶ Arrays

Principio

Todo lo que existe es un objeto, y todo lo que sucede es una llamada a una función - John Chambers
(parafraseado)

Principio

Todo lo que existe es un objeto, y todo lo que sucede es una llamada a una función - John Chambers
(parafraseado)

```
f1 <- function(x) x^2
```


Principio

Todo lo que existe es un objeto, y todo lo que sucede es una llamada a una función - John Chambers
(parafraseado)

```
f1 <- function(x) x^2
```

```
f1(10)
```

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

Partes de las funciones en R

- ▶ `body()`

Partes de las funciones en R

► `body()`

```
body(f1)
```

```
## x^2
```

Partes de las funciones en R

- ▶ `body()`

```
body(f1)
```

```
## x^2
```

- ▶ `formals()`

Partes de las funciones en R

- ▶ `body()`

```
body(f1)
```

```
## x^2
```

- ▶ `formals()`

```
formals(f1)
```

```
## $x
```

La parte menos conocida

La parte menos conocida

- ▶ `environment()`

La parte menos conocida

► `environment()`

```
environment(f1)
```

```
## <environment: R_GlobalEnv>
```


Variables dentro de las funciones

Se pueden declarar *objetos* dentro de las funciones:

```
g1 <- function(x){  
  a <- 20  
  a + 10 + 2*x  
}
```

Al evaluar `g1(7)` se obtiene:

Variables dentro de las funciones

Se pueden declarar *objetos* dentro de las funciones:

```
g1 <- function(x){  
  a <- 20  
  a + 10 + 2*x  
}
```

Al evaluar `g1(7)` se obtiene: 44

Variables dentro de las funciones

Se pueden declarar *objetos* dentro de las funciones:

```
g1 <- function(x){  
  a <- 20  
  a + 10 + 2*x  
}
```

Al evaluar `g1(7)` se obtiene: 44

¿Cuál es el body de `g1`? ¿Y los formals?

Variables dentro de las funciones

Se pueden declarar *objetos* dentro de las funciones:

```
g1 <- function(x){  
  a <- 20  
  a + 10 + 2*x  
}
```

Al evaluar `g1(7)` se obtiene: 44

¿Cuál es el body de `g1`? ¿Y los formals? ¿Qué pasa si se llama `g1` sin ponerle parámetros? i.e. `g1`

Variables externas

Podemos tomar variables del ambiente exterior para llamar funciones:

```
a <- 15  
g2 <- function(b) a + b^2
```

¿Qué valor toma $g2(3)$?

Variables externas

Podemos tomar variables del ambiente exterior para llamar funciones:

```
a <- 15  
g2 <- function(b) a + b^2
```

¿Qué valor toma $g2(3)$? 24

Variables externas

Podemos tomar variables del ambiente exterior para llamar funciones:

```
a <- 15  
g2 <- function(b) a + b^2
```

¿Qué valor toma $g2(3)$? 24

Por lo que al evaluar una función, si no se tiene una variable en el ambiente actual, se busca en el ambiente que está justo superior a este. Y si no se encuentra se vuelve a subir y así sucesivamente. . .

Lazy evaluation

Las funciones no **ocupan** recibir parámetros que no usen. Por lo que hay ciertas funciones en las que si uno no pone parámetros y no se van a usar, entonces no tira error:

Lazy evaluation

Las funciones no **ocupan** recibir parámetros que no usen. Por lo que hay ciertas funciones en las que si uno no pone parámetros y no se van a usar, entonces no tira error:

```
falta <- function(a,b){ a^3 - 2}  
falta(a = 2)
```

```
## [1] 6
```

```
falta(a = 3,b = 2)
```

```
## [1] 25
```

Lazy evaluation

Las funciones no **ocupan** recibir parámetros que no usen. Por lo que hay ciertas funciones en las que si uno no pone parámetros y no se van a usar, entonces no tira error:

```
falta <- function(a,b){ a^3 - 2}  
falta(a = 2)
```

```
## [1] 6
```

```
falta(a = 3,b = 2)
```

```
## [1] 25
```

Sin embargo, si se intenta llamar declarando `b = 2`, sin declarar un valor para `a`, sí va a tirar error.

Lazy evaluation

Las funciones no **ocupan** recibir parámetros que no usen. Por lo que hay ciertas funciones en las que si uno no pone parámetros y no se van a usar, entonces no tira error:

```
falta <- function(a,b){ a^3 - 2}  
falta(a = 2)
```

```
## [1] 6
```

```
falta(a = 3,b = 2)
```

```
## [1] 25
```

Sin embargo, si se intenta llamar declarando `b = 2`, sin declarar un valor para `a`, sí va a tirar error. R tira error en el momento que se percata que ocupa la variable.

Parámetros pre-definidos

Se le puede indicar a una función un valor que debe tomar uno de sus parámetros en caso de que este no sea declarado en la llamada:

```
pre_def <- function(w, x = 3){w^x}  
pre_def(4)
```

```
## [1] 64
```

Parámetros pre-definidos

Se le puede indicar a una función un valor que debe tomar uno de sus parámetros en caso de que este no sea declarado en la llamada:

```
pre_def <- function(w, x = 3){w^x}  
pre_def(4)
```

```
## [1] 64
```

Y si se quiere incluir, se puede incluir:

```
pre_def(4,2)
```

```
## [1] 16
```

Una aclaración

Si la función espera recibir un parámetro, ya se “quema” ese nombre del ambiente actual. Entonces no va a intentar buscarlo más arriba:

```
a <- 1  
func_prueba <- function(a,b){a^2 + b^3}
```

Por lo que si se intenta hacer el llamado: `func_prueba(b = 1)`, se obtiene un **error** pues no se tiene esa variable “definida”

Pregunta

Pregunta

¿Qué cree que pasa si uno intenta llamar una función desde adentro de otra función? ¿Y porqué?

Pregunta

¿Qué cree que pasa si uno intenta llamar una función desde adentro de otra función? ¿Y porqué?

Si llamamos una función desde otra función, recordando que todo en R es un objeto, entonces va a ir a buscar primero al ambiente en el que está la llamada, si no lo encuentra va a ir un ambiente para arriba, y la busca y así sucesivamente hasta que lo encuentre o que se de cuenta que no está definido, en cuyo caso tira un error.

Pregunta

¿Qué cree que pasa si uno intenta llamar una función desde adentro de otra función? ¿Y porqué?

Si llamamos una función desde otra función, recordando que todo en R es un objeto, entonces va a ir a buscar primero al ambiente en el que está la llamada, si no lo encuentra va a ir un ambiente para arriba, y la busca y así sucesivamente hasta que lo encuentre o que se de cuenta que no está definido, en cuyo caso tira un error.

Pero entonces, si el comportamiento de las funciones es tan *flexible* podemos. . .

Funciones dentro de funciones

Yo Dawg'



Figure 1: Meme obligatorio:

Funciones dentro de funciones

```
f2 <- function(x){  
  f3 <- function(y){  
    x+y  
  }  
  f3(5)  
}
```

¿Qué creen que pase en esta función? ¿Cual es el body de f2?
¿Qué pasa si se intenta llamar a f3? ¿Se puede?

Funciones dentro de funciones

```
f2 <- function(x){  
  f3 <- function(y){  
    x+y  
  }  
  f3(5)  
}
```

¿Qué creen que pase en esta función? ¿Cual es el body de f2?
¿Qué pasa si se intenta llamar a f3? ¿Se puede?

```
f2(2)
```

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

```
# f3(1)
```

Funciones dentro de funciones

```
f2 <- function(x){  
  f3 <- function(y){  
    x+y  
  }  
  f3(5)  
}
```

¿Qué creen que pase en esta función? ¿Cual es el body de f2?
¿Qué pasa si se intenta llamar a f3? ¿Se puede?

```
f2(2)
```

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

```
# f3(1)
```

¡Hay mucha flexibilidad con lo que se hace! Pues se pueden definir funciones dentro de otras funciones.

Ambiente de f_3

Ambiente de *f3*

```
f2 <- function(x){  
  f3 <- function(y){  
    x+y  
  }  
  print(environment(f3))  
  # Para imprimir en consola, pero no termina la función  
  f3(5)  
}  
  
f2(2)
```

```
## <environment: 0x000000001504a890>
```

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

Las funciones también son un objeto

Por lo que puede ser parámetros de otras funciones, o incluso el objeto que se obtiene de una función, por ejemplo:

```
fun1 <- function(x){  
  y <- 2  
  function() x - y^2  
}
```

```
fun2 <- fun1(20)
```

¿Cuánto vale ahora fun2()?

Las funciones también son un objeto

Por lo que puede ser parámetros de otras funciones, o incluso el objeto que se obtiene de una función, por ejemplo:

```
fun1 <- function(x){  
  y <- 2  
  function() x - y^2  
}
```

```
fun2 <- fun1(20)
```

¿Cuánto vale ahora fun2()? 16

Las funciones también son un objeto

Por lo que puede ser parámetros de otras funciones, o incluso el objeto que se obtiene de una función, por ejemplo:

```
fun1 <- function(x){  
  y <- 2  
  function() x - y^2  
}
```

```
fun2 <- fun1(20)
```

¿Cuánto vale ahora `fun2()`? 16 El ambiente de `fun2` ya no es el global:

Las funciones también son un objeto

Por lo que puede ser parámetros de otras funciones, o incluso el objeto que se obtiene de una función, por ejemplo:

```
fun1 <- function(x){  
  y <- 2  
  function() x - y^2  
}
```

```
fun2 <- fun1(20)
```

¿Cuánto vale ahora fun2()? 16 El ambiente de fun2 ya no es el global:

```
environment(fun2)
```

```
## <environment: 0x00000000128d2f30>
```

¿Qué cree que pase con el valor de fun2 después del siguiente chunk?

```
y <- 3  
fun2()
```

¿Qué cree que pase con el valor de fun2 después del siguiente chunk?

```
y <- 3  
fun2()
```

```
## [1] 16
```

¿Qué cree que pase con el valor de fun2 después del siguiente chunk?

```
y <- 3  
fun2()
```

```
## [1] 16
```

¿Qué se obtiene como body de fun2? formals? Y si se llama sin ponerle los paréntesis?

Funciones con parámetros como resultado de otra función

Se pueden crear funciones que creen funciones evaluadas en ciertos parámetros, y que luego se puedan evaluar en una variable particular:

```
lineal <- function(a,b){  
  function(x){  
    a*x+b  
  }  
}
```

Funciones con parámetros como resultado de otra función

Se pueden crear funciones que creen funciones evaluadas en ciertos parámetros, y que luego se puedan evaluar en una variable particular:

```
lineal <- function(a,b){  
  function(x){  
    a*x+b  
  }  
}
```

```
recta_1 <- lineal(-1,3)
```

Funciones con parámetros como resultado de otra función

Se pueden crear funciones que creen funciones evaluadas en ciertos parámetros, y que luego se puedan evaluar en una variable particular:

```
lineal <- function(a,b){  
  function(x){  
    a*x+b  
  }  
}
```

```
recta_1 <- lineal(-1,3)
```

```
recta_1(7)
```

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

Ejercicio

Programe una función que evalúe en los parámetros: $\{\beta_0, \beta_1, \beta_2, \eta_1\}$ (que recibe como vector), y retorne la siguiente función

$$Nels(m) = \beta_0 + \beta_1 \exp(-m/\eta_1) + \beta_2 \exp(-m/\eta_1)$$

Ejercicio

Programe una función que evalúe en los parámetros: $\{\beta_0, \beta_1, \beta_2, \eta_1\}$ (que recibe como vector), y retorne la siguiente función

$$Nels(m) = \beta_0 + \beta_1 \exp(-m/\eta_1) + \beta_2 \exp(-m/\eta_1)$$

```
Crea_Nels <- function(parametros){  
  beta0 <- parametros[1]  
  beta1 <- parametros[2]  
  beta2 <- parametros[3]  
  eta1 <- parametros[4]  
  function(m){  
    exp_eta1 <- exp(-m/eta1)  
    beta0 + beta1*exp_eta1 + beta2*exp_eta1  
  }  
}
```

Ejercicio

Se usa así:

```
Nels <- Crea_Nels(c(0.02,0.05,-0.06,1.4))  
Nels(3)
```

```
## [1] 0.01882681
```

Ejercicio

Se usa así:

```
Nels <- Crea_Nels(c(0.02,0.05,-0.06,1.4))  
Nels(3)
```

```
## [1] 0.01882681
```

¿Cuál es el environment de la función Nels?

Ejercicio

Se usa así:

```
Nels <- Crea_Nels(c(0.02,0.05,-0.06,1.4))  
Nels(3)
```

```
## [1] 0.01882681
```

¿Cuál es el environment de la función Nels?

```
environment(Nels)
```

```
## <environment: 0x0000000014f52fe8>
```

¿Qué pasa si se modifica eta1 (como variable)?

Ejercicio

Se usa así:

```
Nels <- Crea_Nels(c(0.02,0.05,-0.06,1.4))  
Nels(3)
```

```
## [1] 0.01882681
```

¿Cuál es el environment de la función Nels?

```
environment(Nels)
```

```
## <environment: 0x0000000014f52fe8>
```

¿Qué pasa si se modifica eta1 (como variable)? ¿Sufre alguna modificación Nels si se llama Crea_Nels otra vez?

Funciones como parámetro:

```
x <- list(1:3,10:15,21:23)  
sapply(x,sum)
```

```
## [1] 6 75 66
```

¿Qué hace sapply?

Funciones como parámetro:

```
x <- list(1:3,10:15,21:23)
sapply(x,sum)
```

```
## [1] 6 75 66
```

¿Qué hace sapply?

```
sapply(x,mean)
```

```
## [1] 2.0 12.5 22.0
```

Es *como* un funcional (460?) recibe 2 objetos, una lista y una función que le aplica a cada entrada de la lista.

Funciones como parámetro:

```
x <- list(1:3,10:15,21:23)
sapply(x,sum)
```

```
## [1] 6 75 66
```

¿Qué hace sapply?

```
sapply(x,mean)
```

```
## [1] 2.0 12.5 22.0
```

Es *como* un funcional (460?) recibe 2 objetos, una lista y una función que le aplica a cada entrada de la lista. Existe toda una familia de funciones de apply, que incluye: lapply, mapply, vapply, tapply, entre otras...

sapply vs lapply

¿Cuál es la diferencia entre el llamado de `sapply(x,sum)` y `lapply(x,sum)`?

sapply vs lapply

¿Cuál es la diferencia entre el llamado de `sapply(x,sum)` y `lapply(x,sum)`?

```
sapply(x,sum)
```

```
## [1] 6 75 66
```

```
lapply(x,sum)
```

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

```
## [1] 6
```

```
##
```

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

```
## [1] 75
```

```
##
```

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

```
## [1] 66
```

Ejemplos de sapply

```
sumo <- function(x,y) x + y  
sapply(1:10,sumo,-10)
```

```
## [1] -9 -8 -7 -6 -5 -4 -3 -2 -1 0
```

Ejemplos de sapply

```
sumo <- function(x,y) x + y  
sapply(1:10,sumo,-10)
```

```
## [1] -9 -8 -7 -6 -5 -4 -3 -2 -1 0
```

```
sapply(10:15,function(y) y^2 -100)
```

```
## [1] 0 21 44 69 96 125
```


sapply con funciones *especiales*

También se puede usar la función `+` que viene pre-definida:

sapply con funciones *especiales*

También se puede usar la función + que viene pre-definida:

```
sapply(1:10, `+`, -10)
```

```
## [1] -9 -8 -7 -6 -5 -4 -3 -2 -1 0
```

```
sapply(1:10, "+", -10)
```

```
## [1] -9 -8 -7 -6 -5 -4 -3 -2 -1 0
```

sapply con funciones *especiales*

Recordando que todo lo que se *hace* es una llamada de una función, se pueden usar las funciones para *accesar* variables, para esto mismo:

```
l1 <- list(x = 1:20,y = -100:-1000,z = letters)
str(l1)
```

```
## List of 3
##  $ x: int [1:20] 1 2 3 4 5 6 7 8 9 10 ...
##  $ y: int [1:901] -100 -101 -102 -103 -104 -105 -106 -107 ...
##  $ z: chr [1:26] "a" "b" "c" "d" ...
```

sapply con funciones especiales

Recordando que todo lo que se *hace* es una llamada de una función, se pueden usar las funciones para *accesar* variables, para esto mismo:

```
l1 <- list(x = 1:20,y = -100:-1000,z = letters)
str(l1)
```

```
## List of 3
##  $ x: int [1:20] 1 2 3 4 5 6 7 8 9 10 ...
##  $ y: int [1:901] -100 -101 -102 -103 -104 -105 -106 -107 ...
##  $ z: chr [1:26] "a" "b" "c" "d" ...
```

```
sapply(l1, `[`,15)
```

```
##      x      y      z
## "15" "-114"  "o"
```

Además, preserva los nombres de la lista, pero lo pasa a vector. Por lo que los números pasan a ser characters.

Ejercicio

Programe una función que recibe 3 parámetros:

- ▶ Un tiempo n , entero positivo
- ▶ Vector de tasas variables futuras
- ▶ Monto que se invierte hoy

La función debe retornar el valor del monto al inicio del año n , reinvirtiéndolo cada año con la tasa que corresponde para ese año.

Para comprobar, utilice las tasas:

$c(0.05, 0.06, 0.045, 0.069, 0.073)$, y el tiempo 3, con un monto de 1200, que le debe dar: 1395.702.

Sugerencia: busque la función `cumprod`

Ejercicio

Programe una función que recibe 3 parámetros:

- ▶ Un tiempo n , entero positivo
- ▶ Vector de tasas variables futuras
- ▶ Monto que se invierte hoy

La función debe retornar el valor del monto al inicio del año n , reinvirtiéndolo cada año con la tasa que corresponde para ese año.

Para comprobar, utilice las tasas:

$c(0.05, 0.06, 0.045, 0.069, 0.073)$, y el tiempo 3, con un monto de 1200, que le debe dar: 1395.702.

Sugerencia: busque la función `cumprod`

Si quisiera llamar esta función con varios tiempos, ¿cómo lo haría? Hágalo con los tiempos de 1 a 3.

Solución:

```
intereses <- c(0.05,0.06,0.045,0.069,0.073)
montosVP <- function(n,intereses,monto){
  cumprod(1+intereses)[n]*monto
}
sapply(1:3,montosVP,intereses,1200)
```

```
## [1] 1260.000 1335.600 1395.702
```

Funciones Infix

Estas funciones son como el +, el −, el *, el %*% (de la tercera tarea, para multiplicar matrices), etc. . . En el sentido de que reciben dos parámetros, uno antes y otro después de escribirlos:

```
`%-` <- function(a,b) paste0('{',a,'-',b,'}')  
`%-`('No','sé')
```

```
## [1] "{No-sé}"
```


Funciones Infix

Estas funciones son como el +, el -, el *, el %*% (de la tercera tarea, para multiplicar matrices), etc. . . En el sentido de que reciben dos parámetros, uno antes y otro después de escribirlos:

```
`%-` <- function(a,b) paste0('{',a,'-',b,'}')  
`%-`('No','sé')
```

```
## [1] "{No-sé}"
```

```
'No' %-% 'sé'
```

```
## [1] "{No-sé}"
```

También se pueden llamar las funciones usuales usando esta primera notación, pero obviamente es más natural usar la segunda:

```
`+`(3.452574,4.700641) == 3.452574 + 4.700641
```

```
## [1] TRUE
```

Esto permite concatenar de forma natural estas operaciones, como cuando uno pone una suma muy larga:

Esto permite concatenar de forma natural estas operaciones, como cuando uno pone una suma muy larga:

```
'Esta' %-% 'frase' %-% 'puede' %-% 'ser' %-% 'muy' %-% 'larga'
```

```
## [1] "{{{{{Esta-frase}-puede}-ser}-muy}-larga}"
```

Nótese que se compone comenzando por el lado izquierdo.

Composición de funciones

Cuando se quieren hacer operaciones un poco más complejas en las cuales se requieran “componer” funciones de forma inmediata, lo que hace R es que va llamándolas por capas comenzando por la primera, y va subiendo hasta llegar al resultado final:

```
sum(floor(log(cumsum(sqrt(1:100)))))
```

```
## [1] 460
```

Composición de funciones

Cuando se quieren hacer operaciones un poco más complejas en las cuales se requieran “componer” funciones de forma inmediata, lo que hace R es que va llamándolas por capas comenzando por la primera, y va subiendo hasta llegar al resultado final:

```
sum(floor(log(cumsum(sqrt(1:100)))))
```

```
## [1] 460
```

Cuando uno programa siempre se tiene que intentar ser lo más **claro** posible. Pues eventualmente alguien va a llegar a leer el código (ya sea la misma persona que lo programó, o alguien más) y va a tener que lidiar con eso (leer con tono despectivo).

Composición de funciones

Cuando se quieren hacer operaciones un poco más complejas en las cuales se requieran “componer” funciones de forma inmediata, lo que hace R es que va llamándolas por capas comenzando por la primera, y va subiendo hasta llegar al resultado final:

```
sum(floor(log(cumsum(sqrt(1:100)))))
```

```
## [1] 460
```

Cuando uno programa siempre se tiene que intentar ser lo más **claro** posible. Pues eventualmente alguien va a llegar a leer el código (ya sea la misma persona que lo programó, o alguien más) y va a tener que lidiar con eso (leer con tono despectivo). Es por esto que...

Composición de funciones

Cuando se quieren hacer operaciones un poco más complejas en las cuales se requieran “componer” funciones de forma inmediata, lo que hace R es que va llamándolas por capas comenzando por la primera, y va subiendo hasta llegar al resultado final:

```
sum(floor(log(cumsum(sqrt(1:100)))))
```

```
## [1] 460
```

Cuando uno programa siempre se tiene que intentar ser lo más **claro** posible. Pues eventualmente alguien va a llegar a leer el código (ya sea la misma persona que lo programó, o alguien más) y va a tener que lidiar con eso (leer con tono despectivo). Es por esto que... vamos a usar nuestra primera librería:

Primera librería

Para instalarlo:

```
install.packages('magrittr')  
# En general es cambiar 'magrittr',  
# por la librería que se quiere
```


Primera librería

Para instalarlo:

```
install.packages('magrittr')  
# En general es cambiar 'magrittr',  
# por la librería que se quiere
```

Para usarlos hay 2 opciones, se pueden acceder las funciones usando `magrittr::funcion_que_voy_a_usar(x,y,z)` (obvio esta no existe),

Primera librería

Para instalarlo:

```
install.packages('magrittr')  
# En general es cambiar 'magrittr',  
# por la librería que se quiere
```

Para usarlos hay 2 opciones, se pueden acceder las funciones usando `magrittr::funcion_que_voy_a_usar(x,y,z)` (obvio esta no existe), o se puede cargar la librería usando el comando `library(magrittr)`, y luego solo se llama la función como cualquier otra `funcion_que_voy_a_usar(x,y,z)`.

Primera librería

Para instalarlo:

```
install.packages('magrittr')  
# En general es cambiar 'magrittr',  
# por la librería que se quiere
```

Para usarlos hay 2 opciones, se pueden acceder las funciones usando `magrittr::funcion_que_voy_a_usar(x,y,z)` (obvio esta no existe), o se puede cargar la librería usando el comando `library(magrittr)`, y luego solo se llama la función como cualquier otra `funcion_que_voy_a_usar(x,y,z)`. En algunas librerías hay información extra sobre las funciones que pone a disposición, se puede acceder usando el comando `vignette('magrittr')`. Y cada objeto debe suele tener su propia página de ayuda dentro de la librería

Pipes:



Figure 2: 'Esto no es una pipa - Magritte'

%>%

Este comando permite evaluar la expresión anterior al comando en la función que sigue. Es decir, $x \%>\% f()$ es equivalente a $f(x)$.

%>%

Este comando permite evaluar la expresión anterior al comando en la función que sigue. Es decir, $x \%>\% f()$ es equivalente a $f(x)$. Por lo que al concatenar funciones, como en el ejemplo anterior, se puede hacer que: $f(g(x))$ sea equivalente a $x \%>\% g() \%>\% f()$. Esto da mayor claridad a la hora de hacer composición de funciones en R:

`%>%`

Este comando permite evaluar la expresión anterior al comando en la función que sigue. Es decir, $x \%>\% f()$ es equivalente a $f(x)$. Por lo que al concatenar funciones, como en el ejemplo anterior, se puede hacer que: $f(g(x))$ sea equivalente a $x \%>\% g() \%>\% f()$. Esto da mayor claridad a la hora de hacer composición de funciones en R:

```
1:100 %>%  
  sqrt() %>%  
  cumsum() %>%  
  log() %>%  
  floor() %>%  
  sum()
```

```
## [1] 460
```

`%>%`

Este comando permite evaluar la expresión anterior al comando en la función que sigue. Es decir, `x %>% f()` es equivalente a `f(x)`. Por lo que al concatenar funciones, como en el ejemplo anterior, se puede hacer que: `f(g(x))` sea equivalente a `x %>% g() %>% f()`. Esto da mayor claridad a la hora de hacer composición de funciones en R:

```
1:100 %>%  
  sqrt() %>%  
  cumsum() %>%  
  log() %>%  
  floor() %>%  
  sum()
```

```
## [1] 460
```

Lo cual es un poco más claro en el orden de operaciones que se realizan, y nos va a ser muy útil cuando veamos dataframes :)

Una función muy importante. . . que ya deberían conocer de progra:

if & else

Una función muy importante... que ya deberían conocer de progra:

if & else

```
Mi_if_1 <- function(x){  
  if(x< 20) 25 else 35  
}  
Mi_if_1(18)
```

```
## [1] 25
```

```
Mi_if_1(21)
```

```
## [1] 35
```

Usando corchetes:

```
Mi_if_2<- function(x){  
  if(x > 18){  
    return(20)  
  } else {  
    return(1:20)  
  }  
}  
Mi_if_2(15)
```

```
## [1] 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17
```

```
Mi_if_2(22)
```

```
## [1] 20
```

Return

Las funciones pueden retornar cualquier clase de objeto, para ser explícitos (y para mayor claridad) se puede usar el comando `return`. Esto es principalmente valioso cuando se quiere evitar un cálculo molesto y largo desde el principio, y se quiere devolver el valor desde el inicio.

ifelse vectorial

Muchas veces se quieren trabajar de forma vectorial los condicionales. Por lo que existe una versión vectorial.

```
Mi_if_3 <- function(x){  
  v1 <- ifelse(x < 10, 5, 3)  
  return(v1)  
}
```

ifelse vectorial

Muchas veces se quieren trabajar de forma vectorial los condicionales. Por lo que existe una versión vectorial.

```
Mi_if_3 <- function(x){  
  v1 <- ifelse(x< 10, 5, 3)  
  return(v1)  
}
```

Si le mandamos un vector, se obtiene un resultado satisfactorio. Mientras las otras funciones devuelven algún tipo de error

```
Mi_if_3(1:15)
```

```
## [1] 5 5 5 5 5 5 5 5 5 3 3 3 3 3 3
```

Si se intenta el llamado anterior con un if normal, da un error:

```
abc <- if(1:15 < 10) 5 else 3
```

for

Algunas veces las funciones vectoriales y los applys no son suficientes para las tareas que se tienen que desempeñar. Por lo que R, también cuenta con for

```
for(i in 1:12) print(i)
```

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

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

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

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

```
## [1] 5
```

```
## [1] 6
```

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

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

```
## [1] 9
```

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

```
## [1] 11
```

```
## [1] 12
```

for

Algunas veces las funciones vectoriales y los applys no son suficientes para las tareas que se tienen que desempeñar. Por lo que R, también cuenta con for

```
for(i in 1:12) print(i)
```

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

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

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

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

```
## [1] 5
```

```
## [1] 6
```

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

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

```
## [1] 9
```

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

```
## [1] 11
```

```
## [1] 12
```


Otro for

```
mi_iterador <- function(vector1){  
  # l1 <- length(vector1)  
  v1 <- vector(length = length(vector1),mode = 'numeric')  
  # rep(0,length(vector1))  
  for(num in 1:(length(vector1) - 1 )){  
    v1[num] <- vector1[num+1] / vector1[num]  
  }  
  return(v1)  
}
```

Otro for

```
mi_iterador <- function(vector1){  
  # l1 <- length(vector1)  
  v1 <- vector(length = length(vector1),mode = 'numeric')  
  # rep(0,length(vector1))  
  for(num in 1:(length(vector1) - 1 )){  
    v1[num] <- vector1[num+1] / vector1[num]  
  }  
  return(v1)  
}
```

```
mi_iterador(seq(from = 12, to = 1400,by = 347))
```

```
## [1] 29.916667  1.966574  1.491501  1.329535  0.000000
```

Ejercicio

Ajuste la función que hizo de `montosVP`, de forma tal que si recibe un n mayor a la longitud del vector que se tiene, devuelva un valor de 0.

Ejercicio

Ajuste la función que hizo de `montosVP`, de forma tal que si recibe un n mayor a la longitud del vector que se tiene, devuelva un valor de 0.

Para realmente terminar de apreciar el valor de las funciones vectoriales, programe la misma función, pero esta vez utilizando un `for`.

Funciones de reemplazo

Estas funciones modifican uno de sus argumentos, y le *reemplazan* el valor de alguna de sus características:

Funciones de reemplazo

Estas funciones modifican uno de sus argumentos, y le *reemplazan* el valor de alguna de sus características:

```
v1 <- 1:12  
names(v1)
```

```
## NULL
```

```
names(v1) <- letters[1:12] # las primeras 12 letras
```

Funciones de reemplazo

Estas funciones modifican uno de sus argumentos, y le *reemplazan* el valor de alguna de sus características:

```
v1 <- 1:12  
names(v1)
```

```
## NULL
```

```
names(v1) <- letters[1:12] # las primeras 12 letras
```

```
names(v1)
```

```
## [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l"
```

Definiendo funciones de reemplazo

Para definir estas funciones, se debe usar la sintaxis 'xxx<-' (sin espacios!) y luego definirla, y luego retornarla con la modificación. Por ejemplo:

```
`tercera<-` <- function(x,value){  
  x[3] <- value  
  x  
}
```

El parámetro de value debe tener ese nombre, pues de lo contrario da un error a la hora de llamarlo.

Definiendo funciones de reemplazo

Para definir estas funciones, se debe usar la sintaxis 'xxx<-' (sin espacios!) y luego definirla, y luego retornarla con la modificación. Por ejemplo:

```
`tercera<-` <- function(x,value){  
  x[3] <- value  
  x  
}
```

El parámetro de value debe tener ese nombre, pues de lo contrario da un error a la hora de llamarlo.

Y para llamarla, se usa la misma sintaxis que con names

```
v1 <- 1:7  
tercera(v1) <- 15075  
v1
```

```
## [1]      1      2 15075      4      5      6      7
```

Definiendo funciones de reemplazo

Para definir estas funciones, se debe usar la sintaxis 'xxx<-' (sin espacios!) y luego definirla, y luego retornarla con la modificación. Por ejemplo:

```
`tercera<-` <- function(x,value){  
  x[3] <- value  
  x  
}
```

El parámetro de value debe tener ese nombre, pues de lo contrario da un error a la hora de llamarlo.

Y para llamarla, se usa la misma sintaxis que con names

```
v1 <- 1:7  
tercera(v1) <- 15075  
v1
```

```
## [1]      1      2 15075      4      5      6      7
```

Reemplazo con entrada

Si se quiere tener más flexibilidad con la definición del reemplazo se puede incluyendo más parámetros. Los cuales deben estar entre el parámetro `x` y el `value` en la definición.

```
`modifica<-` <- function(x,entrada,value){  
  x[entrada] <- value  
  x  
}
```

Reemplazo con entrada

Si se quiere tener más flexibilidad con la definición del reemplazo se puede incluyendo más parámetros. Los cuales deben estar entre el parámetro `x` y el `value` en la definición.

```
`modifica<-` <- function(x,entrada,value){  
  x[entrada] <- value  
  x  
}
```

```
modifica(v1,4) <- 49.7  
v1
```

```
## [1]      1.0      2.0 15075.0    49.7      5.0      6.0
```

while

Algunas veces se quiere iterar hasta que se cumpla cierta condición, por lo que se tiene la función de *while*:

```
a <- -15.3
b <- 7
while(a<b){
  a <- a+2
  b <- b+1
}

print(c(a,b)) # imprime el vector
```

```
## [1] 30.7 30.0
```

Que si se quisiera incorporar dentro de una función se podría.

while

Algunas veces se quiere iterar hasta que se cumpla cierta condición, por lo que se tiene la función de *while*:

```
a <- -15.3
b <- 7
while(a<b){
  a <- a+2
  b <- b+1
}

print(c(a,b)) # imprime el vector
```

```
## [1] 30.7 30.0
```

Que si se quisiera incorporar dentro de una función se podría. Y como en Java, hay que tener cuidado con los ciclos infinitos... Por ejemplo, si se pone en el while de arriba `b <- b+2`, se itera hasta el infinito.

switch

Otra función bastante importante, y útil, es la función *switch*, que funciona como el select que vimos en **VBA**:

```
switch(2,3,4)
```

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

```
switch(7, 'No', 'no', 'tampoco', 'puede', 'pasar',  
        'casi', 'Si!!!', 'te pasaste', 'demasiado')
```

```
## [1] "Si!!!"
```

Solo que toma el numero de entrada que le corresponda. Pero también se le podrían poner nombres:

```
switch('A', B = 'Valor B', A = 'Valor A', C = 'Valor C')
```

```
## [1] "Valor A"
```

Switch con funciones:

Los switch, esperan devolver un objeto, el cual puede ser una función, por lo que luego se puede operar sobre este:

```
switch('X',X = mean,Y = sum,Z = max)(1:15)
```

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


Switch con funciones:

Los switch, esperan devolver un objeto, el cual puede ser una función, por lo que luego se puede operar sobre este:

```
switch('X',X = mean,Y = sum,Z = max)(1:15)
```

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

Si no se hubiera puesto el vector 1:15, se obtiene la función que corresponde.

Switch con funciones:

Los switch, esperan devolver un objeto, el cual puede ser una función, por lo que luego se puede operar sobre este:

```
switch('X',X = mean,Y = sum,Z = max)(1:15)
```

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

Si no se hubiera puesto el vector 1:15, se obtiene la función que corresponde.Haga el llamado pero sin el vector 1:15.

Próximamente:

DataFrames :)