R LECTURES

15/04/2019

REPASO DE LA CLASE ANTERIOR

OPERADORES LÓGICOS

Son operadores que permiten comparar dos enunciados y evaluan a resultado lógico.

- `>` , `>=`
- `<` , `<=`
- `!=` , `==`

Más los operadores && (AND) y || (OR) para elaborar enunciados más complejos

Combinando operadores construimos expresiones condicionales, que R evalúa a `TRUE` o `FALSE` (o `NA`).

EJECUCIÓN CONDICIONAL

```
if (condición) {
  # código que se ejecuta cuando la condición evalua a TRUE
} else {
  # código que se ejecuta cuando la condición evalua a FALSE
}
```

Nota: Si el cuerpo del `if()` tiene una sola línea, podemos obviar los {}'s.

```
if (this) {
  # do that
} else if (that) {
  # do something else
} else {
  #
}
```

FUNCIONES LÓGICAS ACCESORIAS

- `any()` # devuelve `TRUE` si alguno `TRUE`
- `all()` # devuelve `FALSE` si alguno `FALSE`
- `is.na()`, `is.null()` y el resto de la familia `is./algo/()`
- `%in%` # está `x` en este vector?`
- `which()` # devuelve posiciones de elementos `TRUE`
- `identical()` # por ej., *numeric* vs. *integer*
- muchas otras.

COMPONENTES DE UN `FOR`

Loops son bucles y se usan para repetir código.

- Output: siempre es buena idea crear el objeto antes de calcularlo.
- Secuencia: variable sobre la que funciona el bucle.

```
for (x in xs)
for(i in seq_along(df))
for (nm in names(xs))
```

- Cuerpo: código que se ejecuta las veces que la secuencia indique.

`while()` es un loop controlado por una expresión condicional.

Nota: Si el cuerpo de `for()` o del `while()` tienen una sola línea, podemos obviar las {}'s.

ANATOMÍA DE F()

```
# mi función se llama alta_funcion, con dos argumentos
alta_funcion <- function(arg1 = 10, arg2 = TRUE, ...){

  # acá empieza mi código
  library(paquete_externo)
  x <- funcion_externa(arg_ext = arg1)
  ...
  alto código
  código y más código
  ...
  alto_resultado <- mansa_funcion(arg2) # genero alto_resultado

  return(alto_resultado) # devuelvo alto_resultado
}
```

```
# llamo a mi función de distintas maneras
x_default <- alta_funcion() # uso arg1 = 10 y arg2 = TRUE
x_100_F   <- alta_funcion(100, FALSE)
x_200_T   <- alta_funcion(200, TRUE)
mi_var    <- alta_funcion(arg2 = FALSE, arg_ext = 10.2) # uso arg1 = 10
```

SUBSETTING, CON LA FUNCIÓN `SUBSET()`

`?subset()` *Return subsets of vectors, matrices or data frames which meet conditions.*

```
subset(airquality, Temp > 80, select = c(Ozone, Temp))
subset(airquality, Day == 1, select = -Temp)
subset(airquality, select = Ozone:Wind)
```

VECTORIZACIÓN

Se trata de operaciones que aplican a un vector, elemento por elemento.

1. Los pasos se simplifican al no pensar en los elementos del vector, si no en el vector en sí.
2. Los bucles en una función vectorizada están hechos en C y no en R, y por lo tanto son mucho más rápidos.

```
# Sin vectorización:
for (i in 1:length(x)) z[i] <- x[i] + y[i]

# Con vectorización:
z <- x + y
```

- Ref: Vectorise from Advanced R

OPERADORES Y FUNCIONES VECTORIZADOS

1. ==, `&`, y |
2. Corchetito de subsetting: Ej.: `x[is.na(x)] <- 0` donde `x` es vector, matriz o dataframe.
3. Otras funciones: +, –, *, `cumsum()`, `diff()`, `rowSums()`, `colSums()`, `rowMeans()`, `colMeans()`, etc.
4. `any(x == 10)` es mucho más rápido que `10 %in% x`.

```
c(T,T,F,F) == c(T,F,T,F)
[1]  TRUE FALSE FALSE  TRUE
c(T,T,F,F) & c(T,F,T,F)
[1]  TRUE FALSE FALSE FALSE
c(T,T,F,F) | c(T,F,T,F)
[1]  TRUE  TRUE  TRUE FALSE
```

OPERADORES LÓGICOS SIMPLES O DOBLES

- Los operadores simples `&` (AND) y `|` (OR) son vectorizados
- Los operadores dobles `&&` (AND) y `||` (OR) evalúan de izquierda a derecha solo el primer elemento. Se procede hasta que el resultado se alcanza.

```
NA & T
[1] NA
NA & F
[1] FALSE
c(NA, T) && c(T, T)
[1] NA
c(NA, T) && c(F, T)
[1] FALSE
c(NA, T) & c(T, T)
[1]   NA TRUE
c(NA, T) & c(F, T)
[1] FALSE  TRUE
```

EFICIENCIA

```
x <- runif(1000000)
y <- runif(1000000)
z <- vector(length=1000000)

system.time(z <- x + y)
  user  system elapsed
  0.052   0.016   0.068

system.time(for (i in 1:length(x)) z[i] <- x[i] + y[i])
 user  system elapsed
 8.088   0.044   8.175
```

FUNCIONALES - FAMILIA *APPLY()

`lapply()`, `sapply()`, `apply()` y `tapply()` (hay más...)

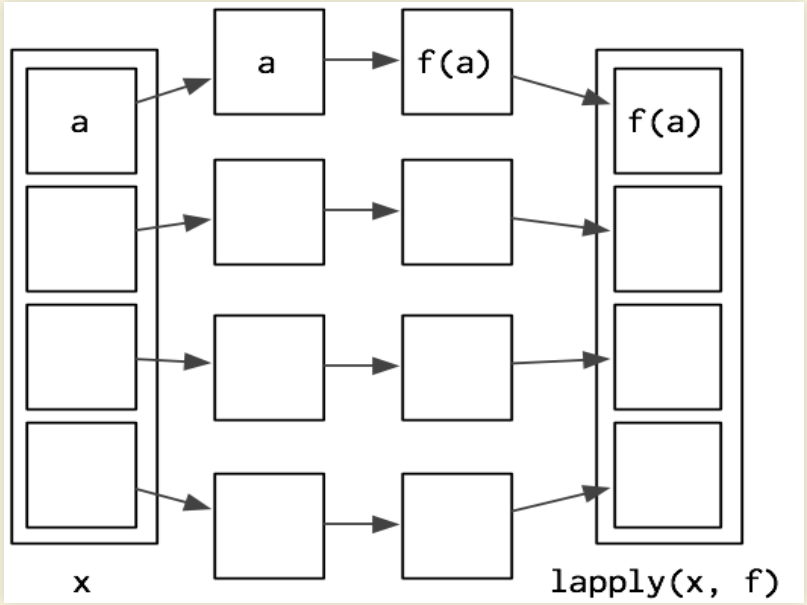
Combinan algo similar a vectorización, pero con funciones. Suelen ser la respuesta a "¿Cómo hago para procesar (por ejemplo transformar o extraer información de) cada elemento de este vector?"

Hemos hecho cosas similares con `summarise()` + `group_by()` y `mutate()`.

Ref: ver también el paquete plyr, esta web y este paper

LAPPLY( )

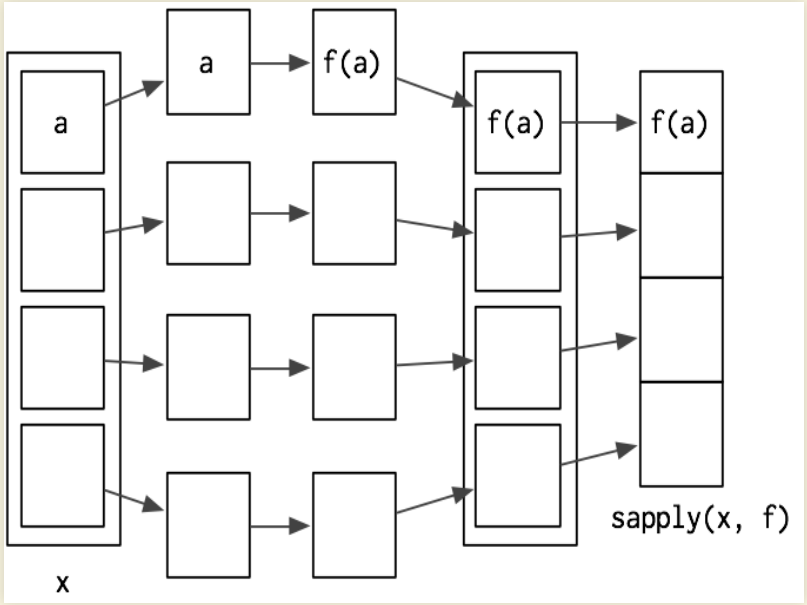
LLama a una función especificada en cada componente de una lista y devuelve otra lista.




```
lapply(list(1:3,25:29), median)
[[1]]
[1] 2
[[2]]
[1] 27
```

SAPPLY()

En algunos casos, la lista que devuelve `lapply()` puede ser simplificada a un vector o a una matriz. Esto es justo lo que hace `sapply()`.




```
sapply(list(1:3,25:29),median)
[1]  2 27
```

`lapply()` y `sapply()` en data frames

Tanto `lapply` como `sapply` están pensados para listas, por lo tanto funcionan bien en *data.frames*.

```
lapply(economics, mean)
$date
[1] "1991-05-17"

$pce
[1] 4843.51

$pop
[1] 257189.4

$psavert
[1] 7.936585

$uempmed
[1] 8.610105

$unemploy
[1] 7771.557

sapply(economics, mean)
        date          pce          pop      psavert      uempmed     unemploy
```

APPLY()

`apply()` es para matrices (o sea, objetos tipo *matrix*). Tienen la particularidad que podemos elegir, con un argumento, aplicar una función a filas o a columnas.

`apply(m,dimcode,f,fargs)`

- m: matriz
- dimcode: 1 o 2, 1 se aplicamos a filas, 2 a columnas.
- f: función que vamos a aplicar.
- fargs: argumentos adicionales.

```
z
[,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6

apply(z,2,mean)
[1] 2 5
```

APPLY( ) - UN EJEMPLO CON FUNCIÓN PROPIA

```
z
[,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
f <- function(x) x/c(2,8)
y <- apply(z,1,f)
y
[,1] [,2] [,3]
[1,]  0.5 1.000 1.50
[2,]  0.5 0.625 0.75
```

TAPPLY( )

para aplicar funciones a vectores con factores. Similar a `group_by` + `summarise`, pero solo con *factores*.

```
ages <- c(25,26,55,37,21,42)
affils <- c("R","D","D","R","U","D")
tapply(ages,affils,mean)
D   R   U
41  31  21
```

```
d <- data.frame(list(gender=c("M","M","F","M","F","F"),
+ age=c(47,59,21,32,33,24),income=c(55000,88000,32450,76500,123000,45650)))
d
  gender age income
1 M      47  55000
2 M      5 9 88000
...
d$over25 <- ifelse(d$age > 25,1,0)
d
  gender age income over25
1 M      47  55000  1
2 M      59  88000  1
...
tapply(d$income,list(d$gender,d$over25),mean)
0       1
F 39050 123000.00
M NA    73166.67
```

PRÁCTICA 8

Descargar práctica 8.