Fundamentos de programación I

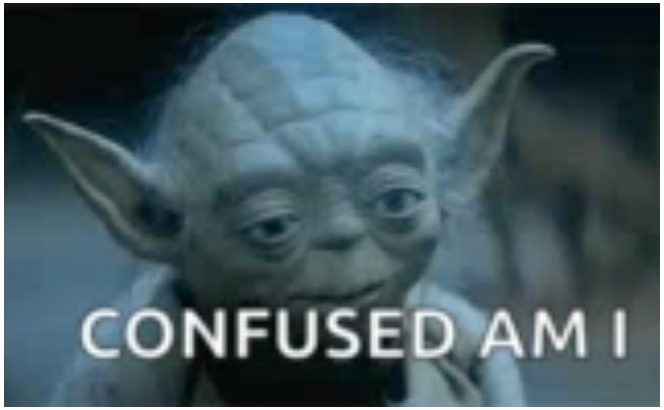
Estructuras de control en R

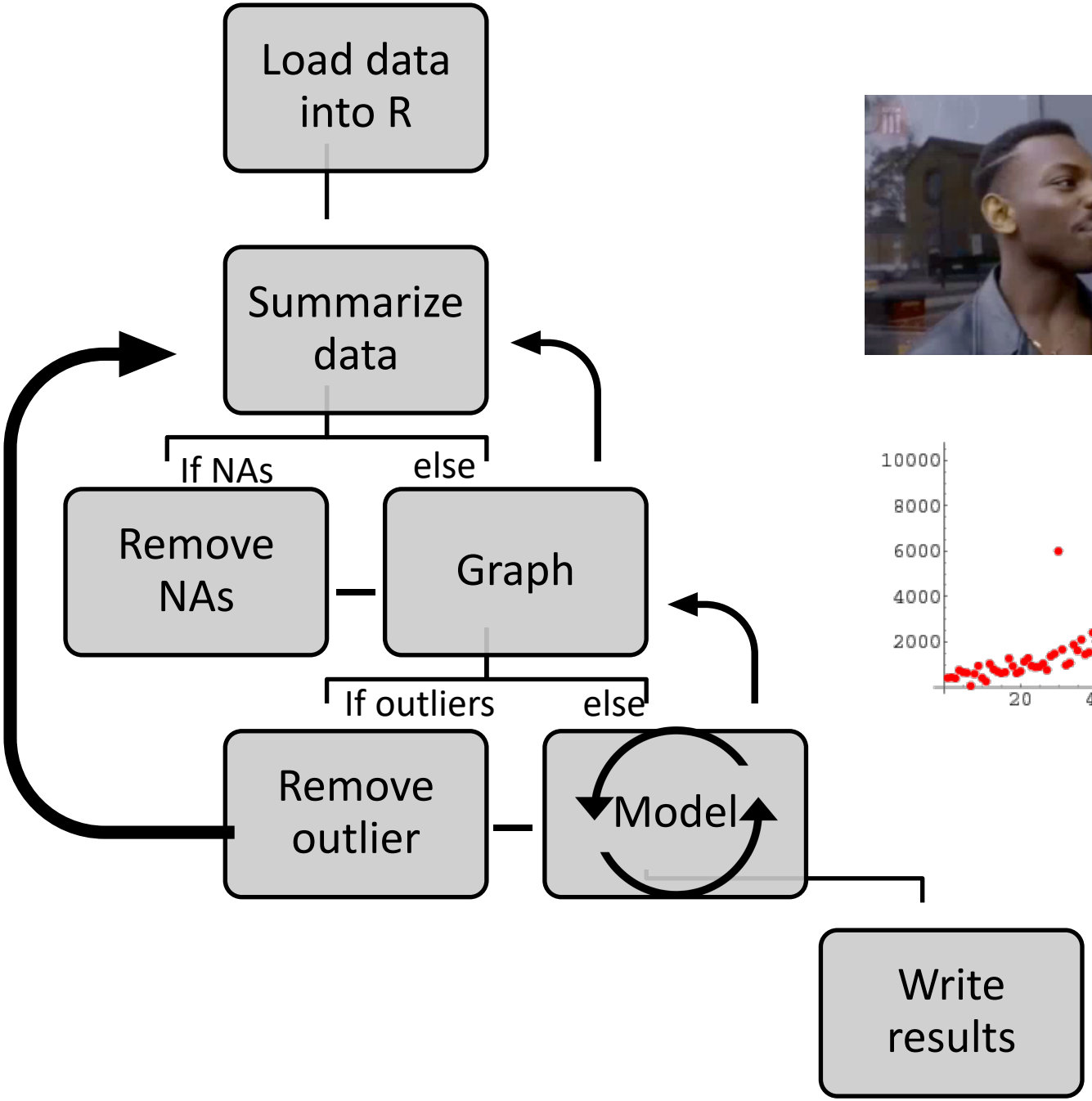
- Bucles
- Funciones

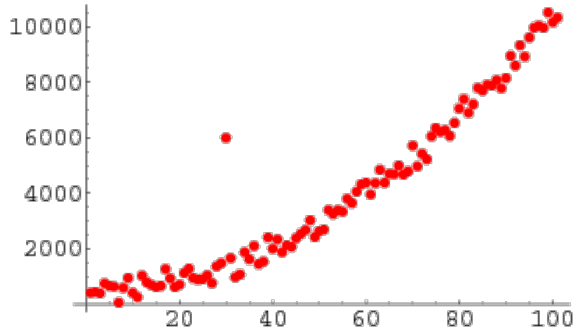
Que son las estructuras de control?

Las estructuras de control son bloques de código que determinan cómo se ejecutan otras secciones de código.

¡Ellas te ayudan a controlar como fluye tu código!

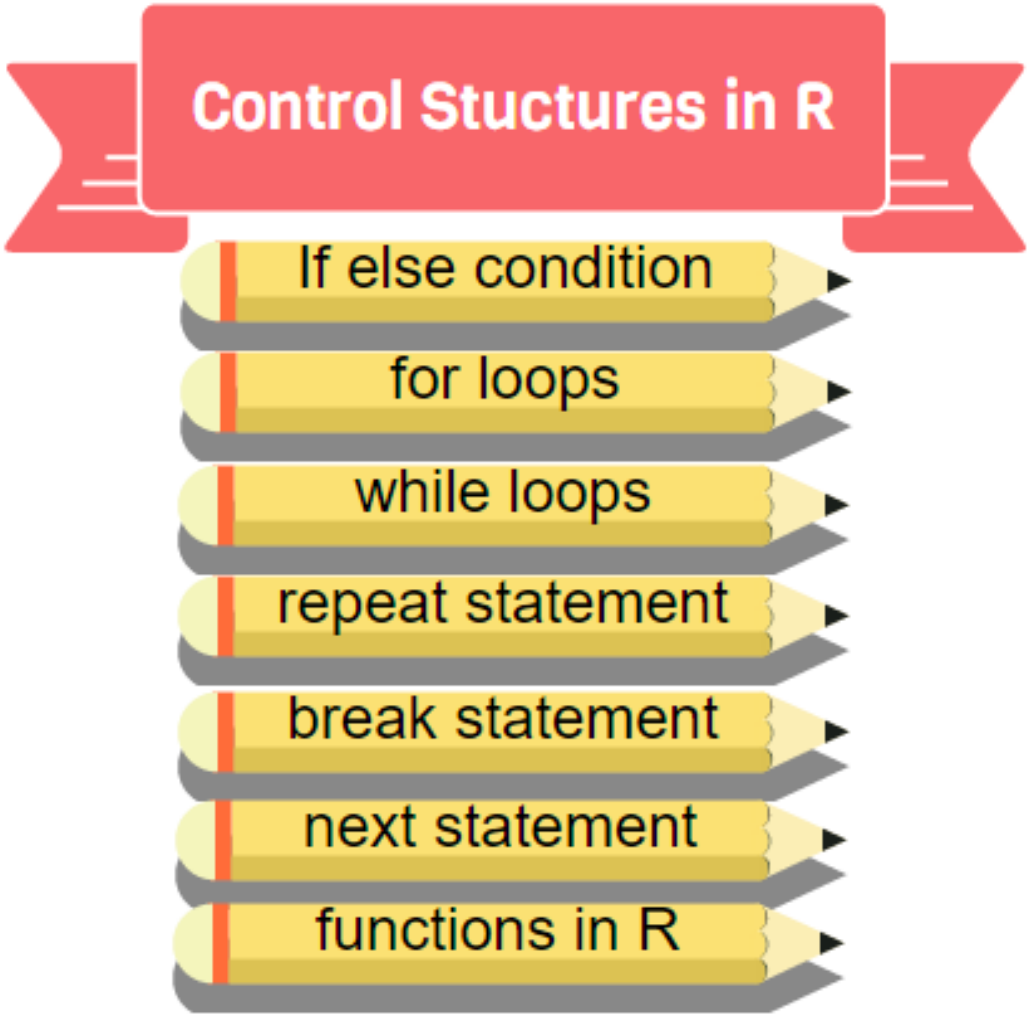



Load data into R
Summarize data
If NAs
else
Remove NAs
Graph
If outliers
else
Remove outlier
Model
Write results


10000
8000
6000
4000
2000
20
40
60
80
100

Estructuras de control

¡Implementan la toma de decisiones sobre qué hacer a continuación!



if (si)


```
if (condición) {
    #codigo
}
```

Solo completa el código en { } **if** se cumplen las condiciones

```
if (x==5) {codigo}
if (x==5 & y==7) {codigo}
if (x==5 | y>=17) {codigo}
if (1 %in% c(1,2,3,5)) {codigo}
```

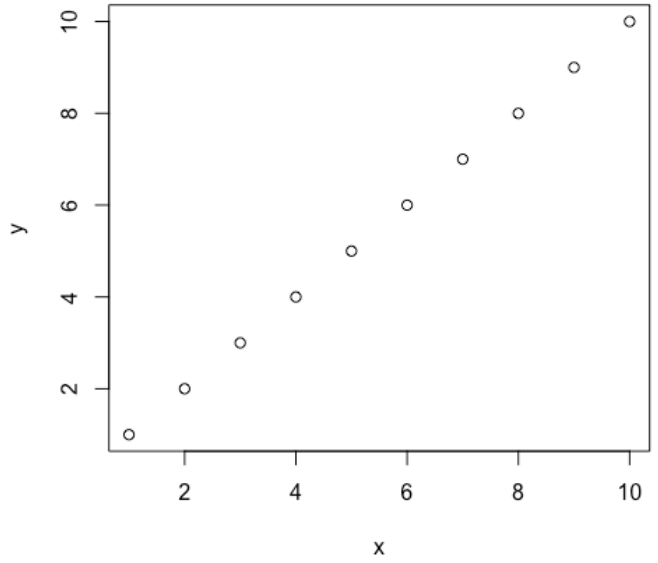
If

Solo grafique x vs y si sus longitudes son las mismas.

```
x=1:10
y=1:11

if (length(x) == length(y)) {
      plot(x,y)
}
```

If ejemplo

```
x=1:10
y=1:10

if (length(x) == length(y))
      plot(x,y)
      }
```



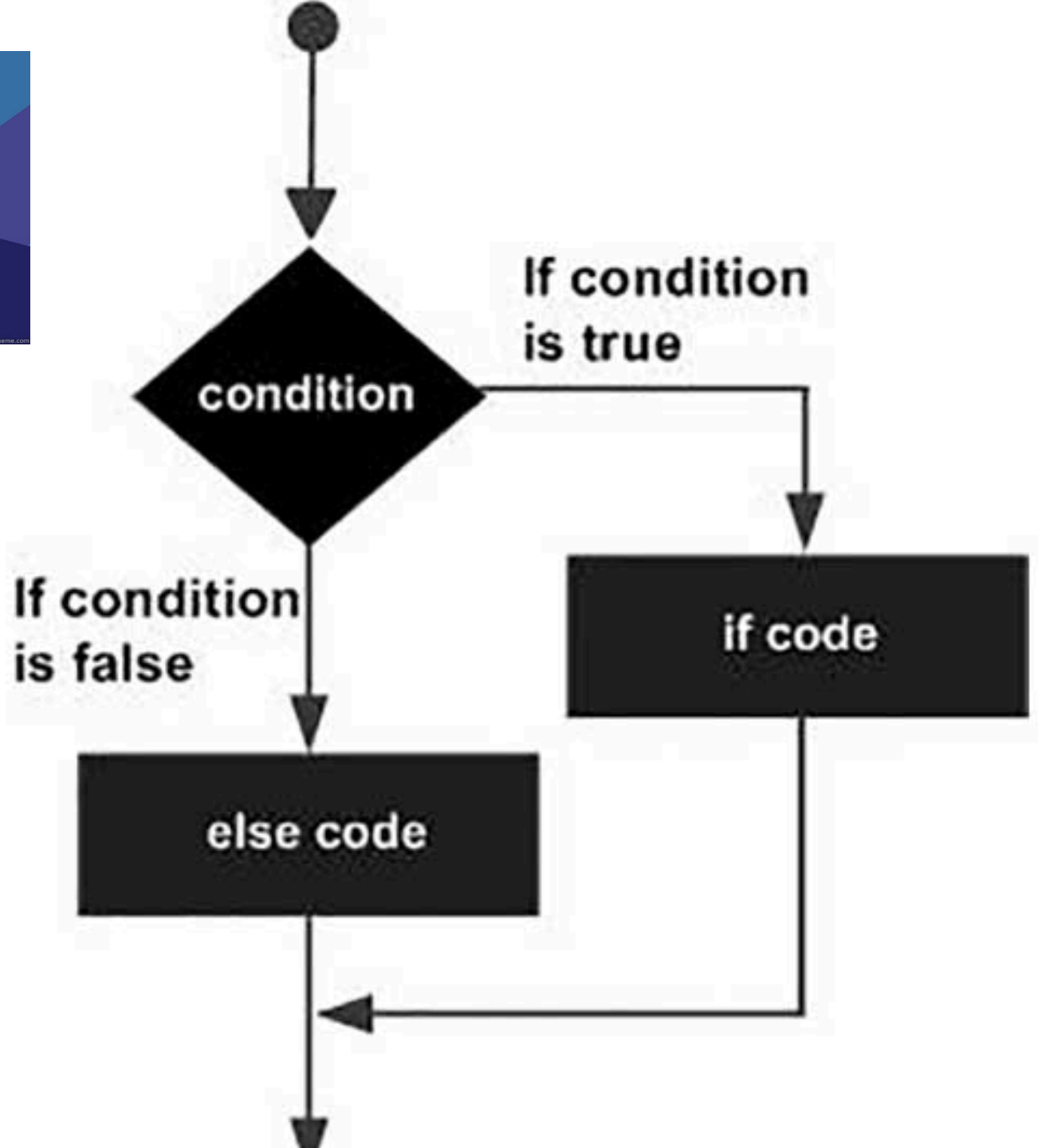
Error in plot.new() : figure margins too large

If/else

Si desea que R elija diferentes rutas según la condición que se cumpla, use una declaración if / else.


```
if(condición) {
    #haga esto
    } else {
    #haga otra cosa
}
```


I SAID DO IT...
OR ELSE.
quickmeme.com

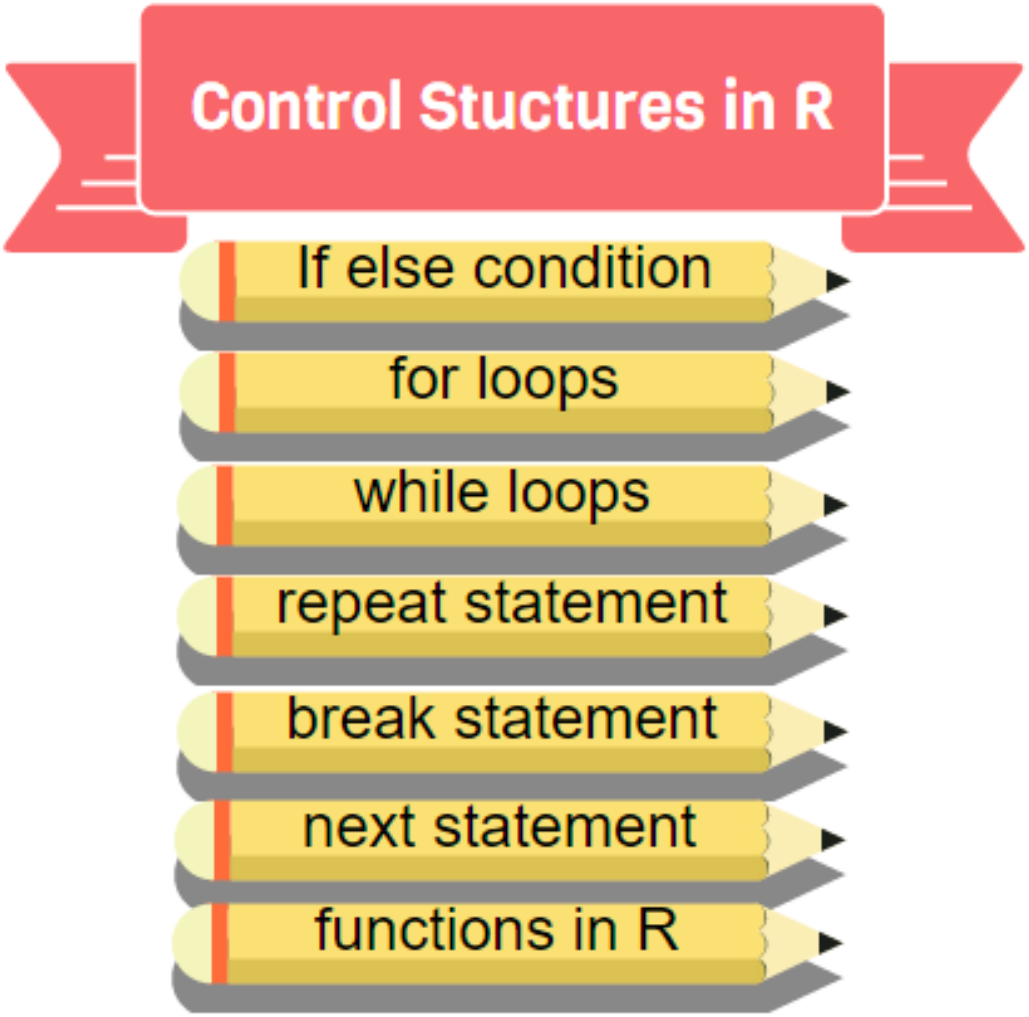

condition
If condition is true
If condition is false
if code
else code

If/else statements

Solo grafique x vs y si sus longitudes son las mismas.

```
x=1:10
y=1:11

if (length(x) == length(y)) {
      plot(x,y)
   } else{
     print("AH! Las longitudes son diferentes")
   }
```

```
[1] "AH! Las longitudes son diferentes"
```

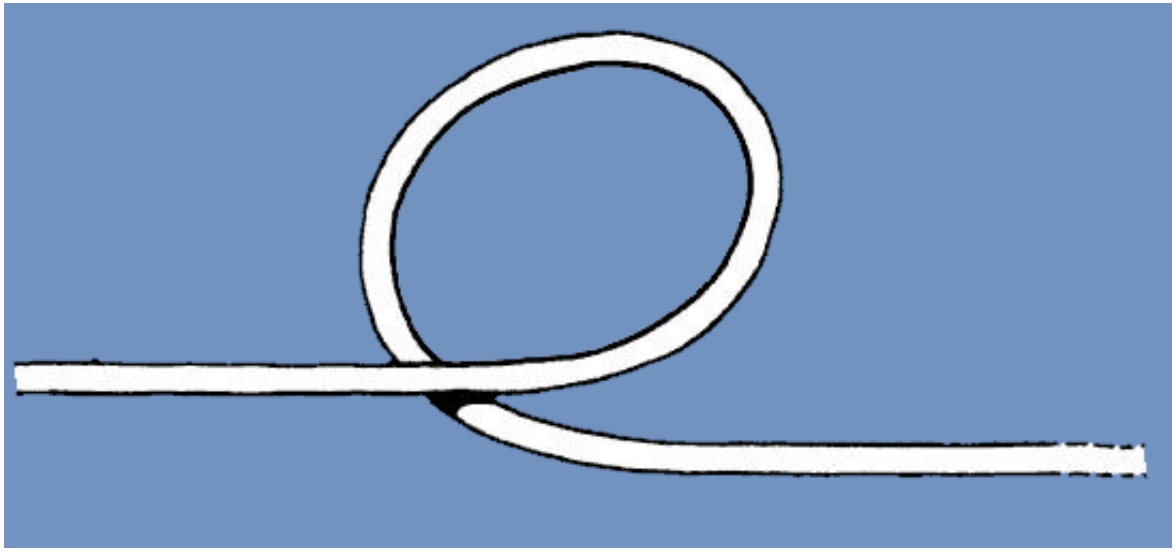

Control Stuctures in R
If else condition
for loops
while loops
repeat statement
break statement
next statement
functions in R

Bucles (loops)

for

Una estructura de control para recorrer en bucle y repetir una tarea!

for loops le permiten repetir el código una y otra vez, cambiando las cosas ligeramente cada vez

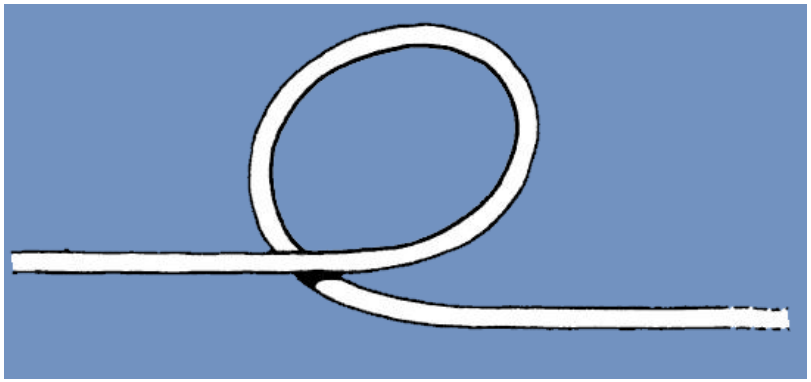


Ejemplos:

Hacer un histograma de cada columna

Tome la suma de cada columna y grafíquelo

for loops



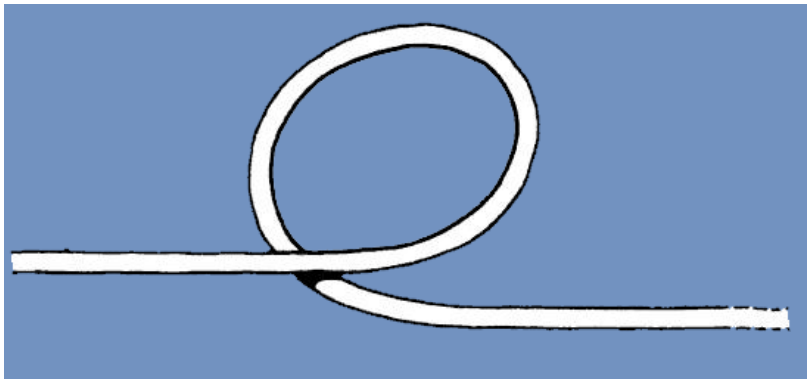
```
for (i in Secuencia) {
    #Haz algo
}
```

Pruebe lo siguiente:

```
for (i in 1:100) {
    print(i)
}
```

`i` es un contador, sin embargo, podrías usar algo más. Convencionalmente usamos i, j, k para contadores

Indentación



- La “indentación” ayuda a la legibilidad, pero también se puede poner todo en una línea

```
for (i in 1:100) {
      print(i)
}
```

…es lo mismo que

```
for (i in 1:100) {print(i)}
```

Ejemplo

```
for (i in c(1,4,5,8)) {
      print(i*2)
}
```

```
[1] 2
[1] 8
[1] 10
[1] 16
```

Imagine que necesita calcular la mean y sd para cada columna en el conjunto de datos mtcars

```
for (i in 1:ncol(mtcars)) {
  #print name of column i
  print(colnames(mtcars)[i])

  #print mean of column i
  print(mean(mtcars[,i]))

  #print sd of column i
  print(sd(mtcars[,i]))

}
```

	mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
Mazda RX4	21.0	6	160	110	3.90	2.620	16.46	0	1	4	4
Mazda RX4 Wag	21.0	6	160	110	3.90	2.875	17.02	0	1	4	4
Datsun 710	22.8	4	108	93	3.85	2.320	18.61	1	1	4	1
Hornet 4 Drive	21.4	6	258	110	3.08	3.215	19.44	1	0	3	1
Hornet Sportabout	18.7	8	360	175	3.15	3.440	17.02	0	0	3	2
Valiant	18.1	6	225	105	2.76	3.460	20.22	1	0	3	1

Looping through species

Imagine you need to calculate the mean and sd for every column in the **mtcars** data set

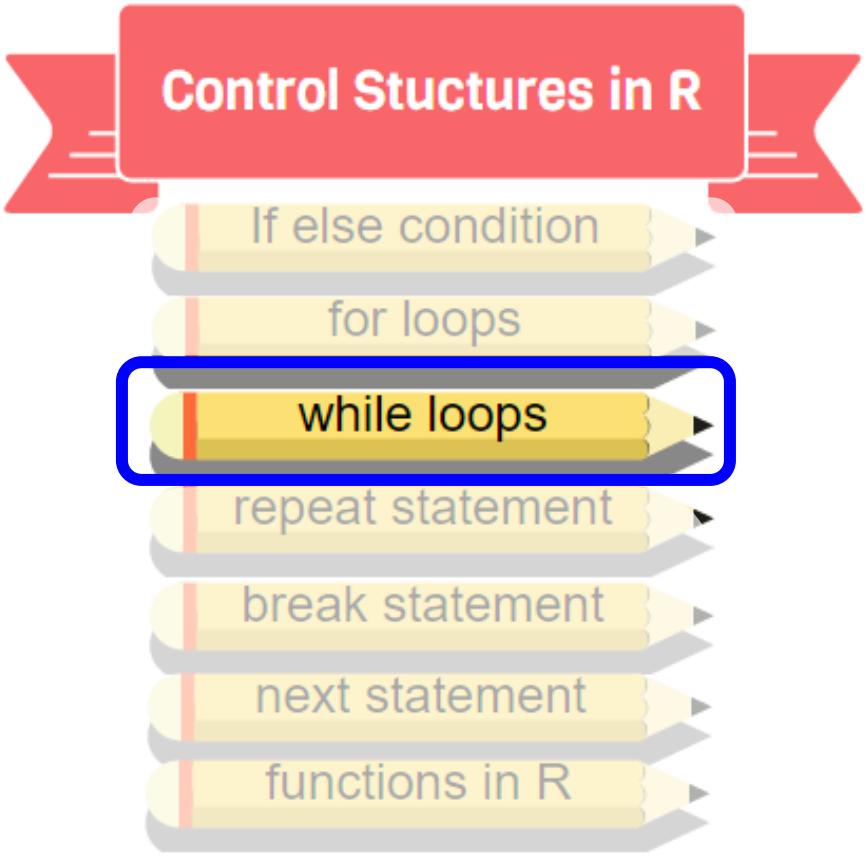
```
for (i in 1:ncol(mtcars)) {
  #print name of column
  print(colnames(mtcars)[i])

  #print mean of column i
  print(mean(mtcars[,i]))

  #print sd of column i
  print(sd(mtcars[,i]))

}
```

	mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
Mazda RX4	21.0	6	160	110	3.90	2.620	16.46	0	1	4	4
Mazda RX4 Wag	21.0	6	160	110	3.90	2.875	17.02	0	1	4	4
Datsun 710	22.8	4	108	93	3.85	2.320	18.61	1	1	4	1
Hornet 4 Drive	21.4	6	258	110	3.08	3.215	19.44	1	0	3	1

Prints to screen:

```
[1] "mpg"
[1] 20.09062
[1] 6.026948
[1] "cyl"
[1] 6.1875
[1] 1.785922
[1] "disp"
[1] 230.7219
[1] 123.9387
[1] "hp"
[1] 146.6875
[1] 68.56287
[1] "drat"
[1] 3.596563
```

Control structures help you **control the flow** of your code!



while

- while ejecuta una serie de instrucciones mientras alguna condición siga siendo verdadera

```
while (condition) {
    #code to run
}
```

Ejemplo

while

```
i <- 1


while (i<6){
  #print the current i
  print(i)
  #add one to i
  i<-i+1
}
```

```
[1] 1
[1] 2
[1] 3
[1] 4
[1] 5
```

While otro ejemplo

Si hay 10.000 osos polares en el año 2015 y están disminuyendo al 9% anual, ¿en qué año caerán por debajo de 500 individuos?

```
N <- 10000
year <- 2015

while (N>500) {
      #reduce the population by 9%
      N <- N*0.91
      #add one to the year
      year <- year+1
   }

year
[1] 2047
```

si terminas en un bucle infinito sin punto de parada, ¡presiona <ESC>para detenerlo!

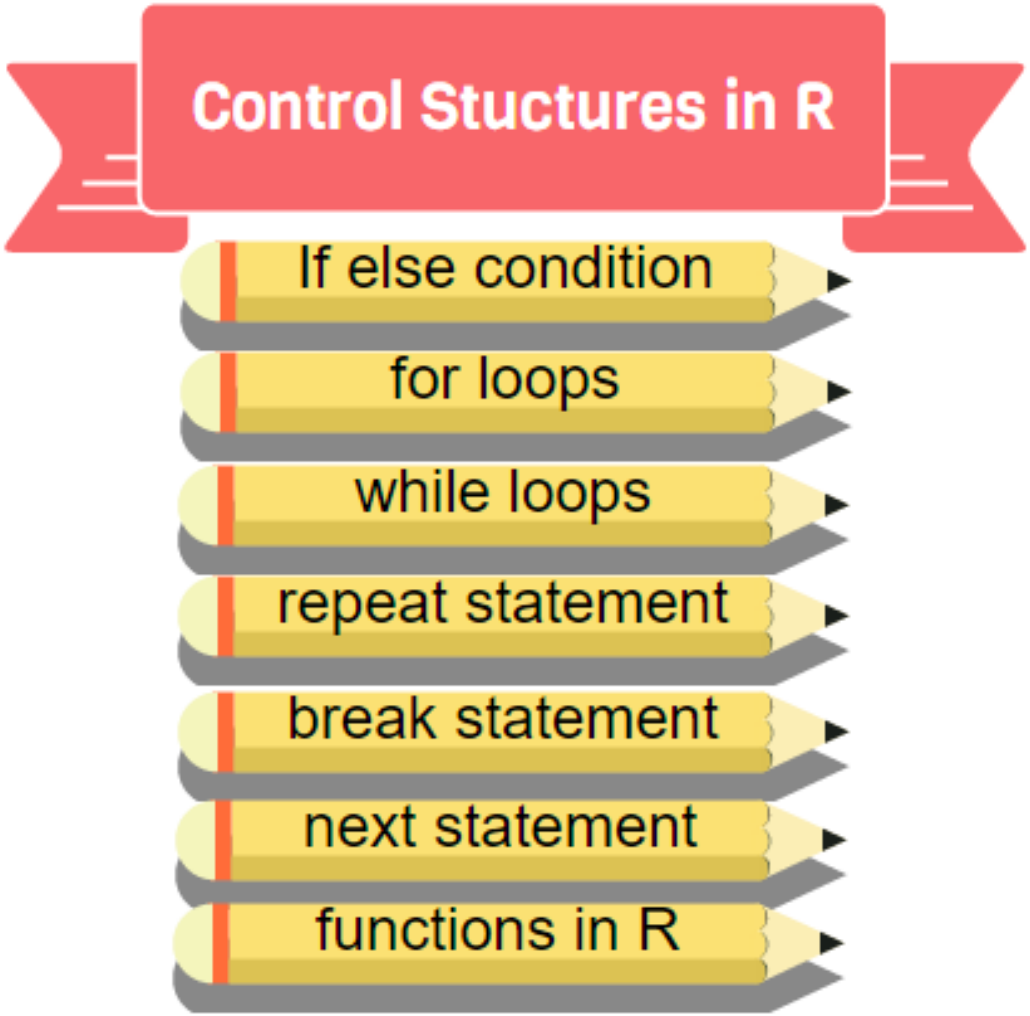
Estructuras de control!

¡Implementan la toma de decisiones sobre qué hacer a continuación!

Pasamos por los más comunes...

¿Quieres saber más?

? Control



Fundamentos de programación II

- **¡Funciones!**

Una mirada más cercana

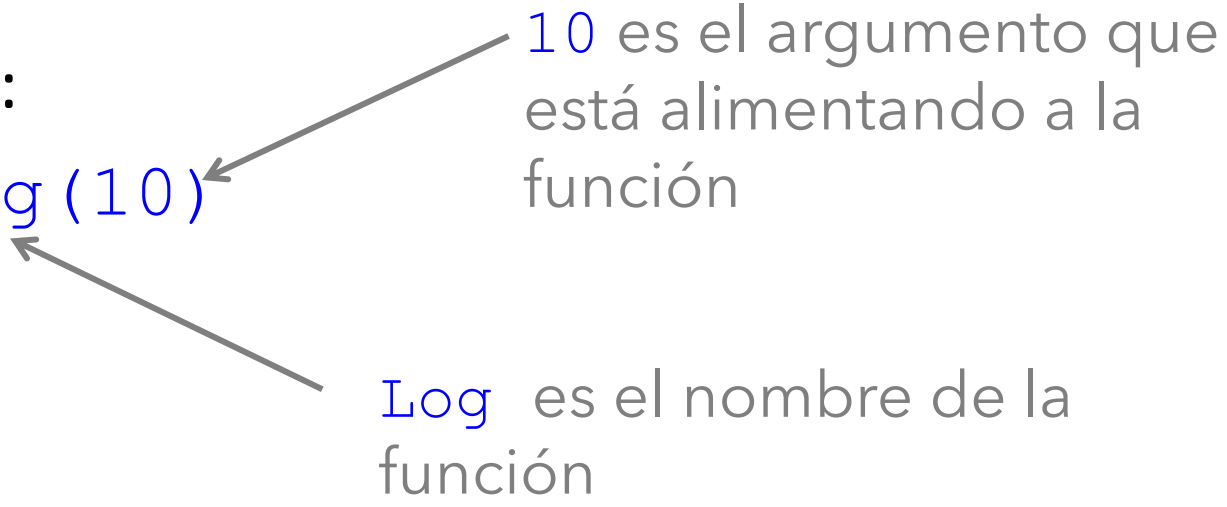
- **Escribir tus propias funciones!**

```
FunctionName(arguments)
```

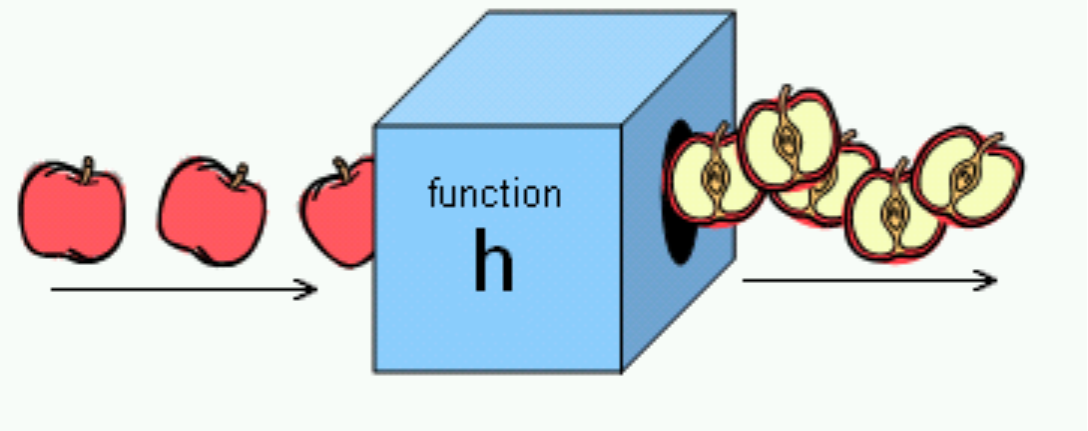
Todo lo que escribamos entre paréntesis se denomina argumentos de la función.

por ejemplo:

```
log(10)
```



Propiedades de las funciones

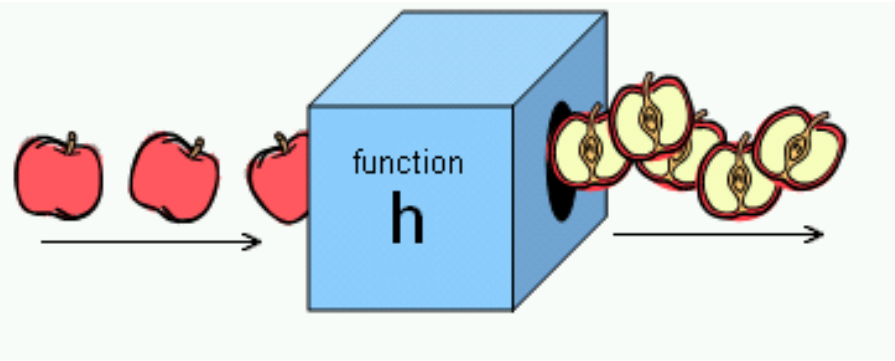


- Tener un nombre
- Se proporcionan los argumentos
- Hacen algo
- Retornan algo para su uso posterior

```
mean() #función que toma el promedio de un vector

plot() # Función genérica para graficar.

sort(x) # Ordena el vector x
```

ver builtins() para una lista de todas las funciones integradas

¡Las funciones simplifican el codigo!



- `sd()`

$$\sqrt{\frac{\sum |x - \bar{x}|^2}{n}}$$

Tambien pueden escribir:

```
sqrt(sum((a-mean(a))^2/(length(a)-1)))
```

– Pero es mucho más fácil usar la función `sd()`

Funciones.. ¡Son sólo un código que alguien escribió!

Funciones... ¿por qué?!

- Cuando R no tenga una función integrada para sus necesidades

- Para evitar la duplicación de código: si tiene varias copias de código casi idéntico, significa que debe colocarlo en una función.

- Es mejor una función (una línea) en lugar de ejecutar muchas líneas de código! ¡Evita errores!

Sintaxis básica para escribir una función

name of function

arguments of function

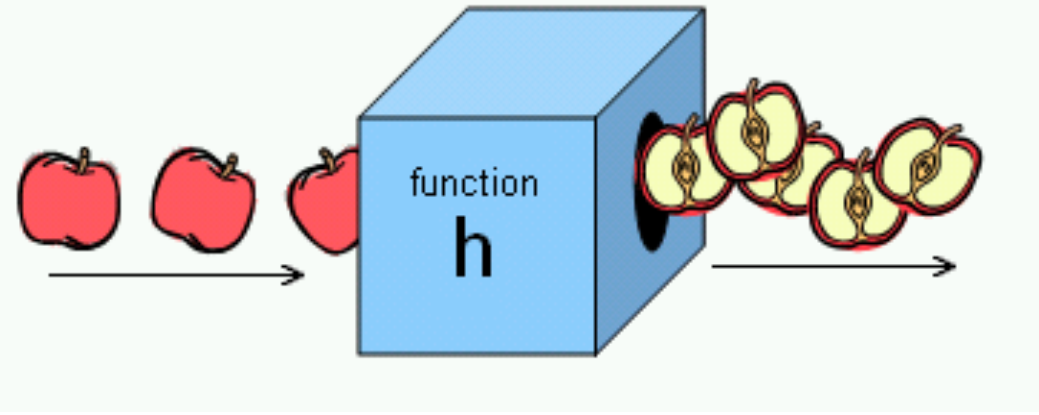
start of the function

```
FunctionName <- function(input) {
      #code that does stuff
      return(output)
}
```

R commands for computation: everything inside { }.

What it should spit out – delimited using return()

end of the function



Una función simple

name of function

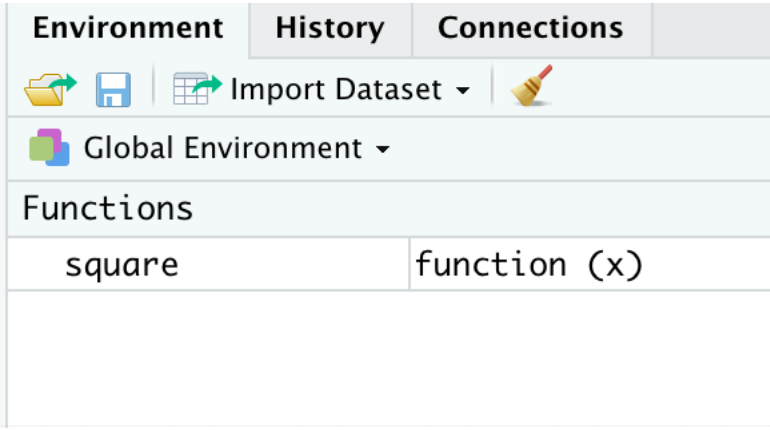
arguments of function (input)

```
square <- function(x) {
        y<-x*x
        return(y)
      }
```

R commands for computation: everything inside { }.

Output

```
square(3)
 9
```



You See Your new R Function in Rstudio!

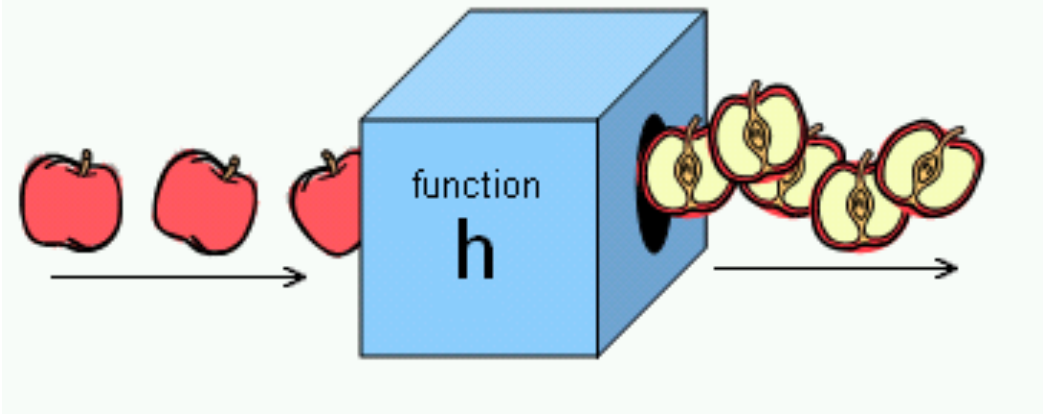
Indentacion y estilo


```
FunctionName <- function(input) {
    #code that does stuff
    return(output)
}
```

más fácil de leer

Ponga cada operación en su propia línea

Poner esto en su propia línea para ver dónde termina la función



En otras palabras,…

Esto es más difícil de leer…

```
square <-function(x){y<-x*x;return(y)}
```

…. Esto es un poco más fácil

```
square <-function(x){
     y<-x*x
     return(y)
}
```

- Los argumentos de las funciones pueden ser obligatorios u opcionales.

`log` toma 2 argumentos
  - `x` (algún número)
  - `base` (la base)

- `?log`

Usage

```
log(x, base = exp(1))
```

- `x` se requiere, pero `base` es opcional (por defecto es exp(1))

```
log(2.718282)
[1] 1

log(2.718282, base=10)
[1] 0.4342945
```

Optional argument

¡ Argumentos opcionales!

- Esta función no tiene argumentos opcionales

```
myfunc <- function(var1, var2)
```

- Para que los argumentos sean opcionales, utilice **=**

```
myfunc <- function(var1=1, var2=0)
```

Para este último caso, todos estos codigos funcionarían:

```
myfunc()
myfunc(1,2)
myfunc(var2=5)
```

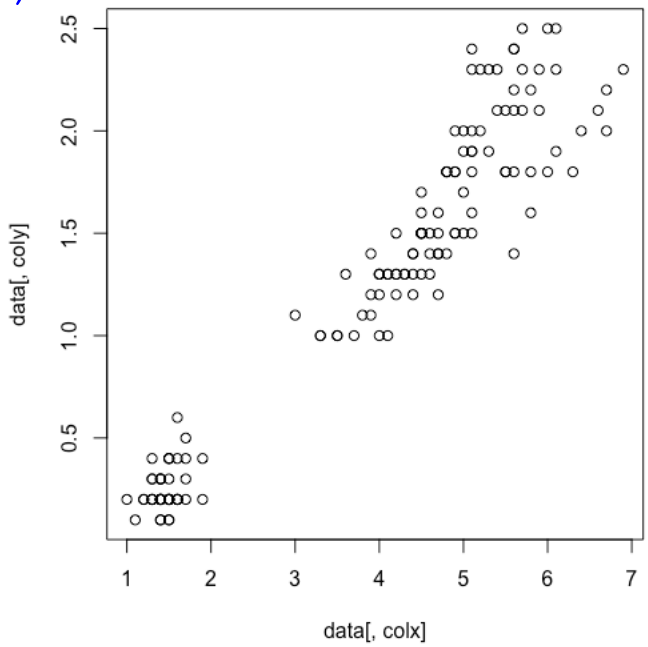
Ejemplo

Supongamos que desea hacer una función que grafique algunas columnas de un data frame

3 argumentos, ninguno opcional

```
plotcols <- function(data, xcol, ycol) {
   plot(data[,xcol],data[,ycol])
}
```

```
plotcols(iris,3,4)
```

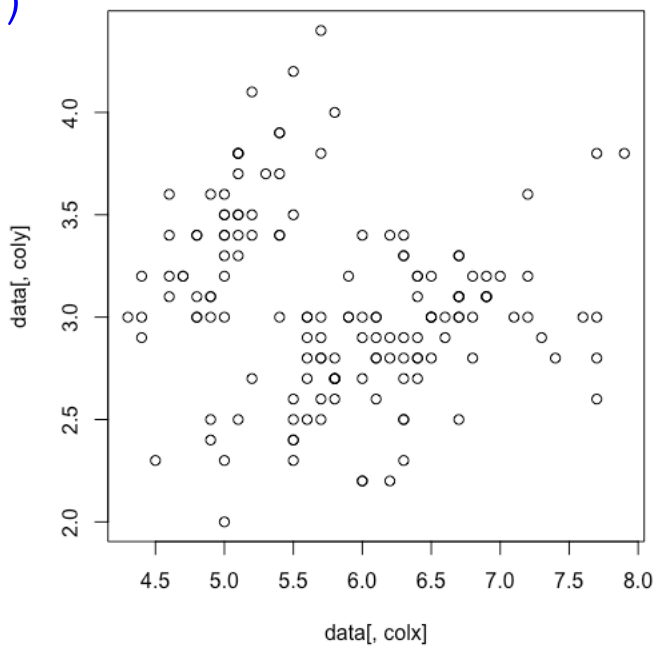


Ejemplo

opcional proporcionando valores predeterminados

```
plotcols <- function(data, xcol=1, ycol=2) {
   plot(data[,xcol],data[,ycol])
}
```

```
plotcols(iris)
```



A menudo necesitamos mas de un valor en una función

```
compute.stat <- function(xvec) {
  ave <- mean(xvec)
  var <- var(xvec)
  sd  <- sqrt(var)
  out <- c(ave=ave, var=var, sd=sd)
  return(out)
}
```

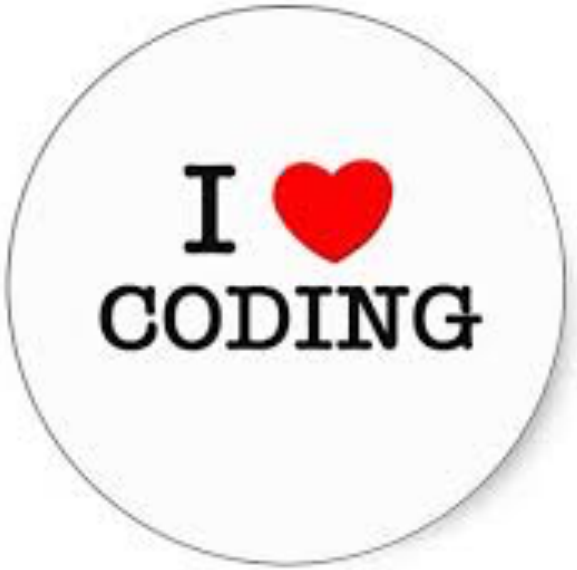
Fundamentos de programación III

Funciones:

apply!

Acelerar el Código

Buenas prácticas de código

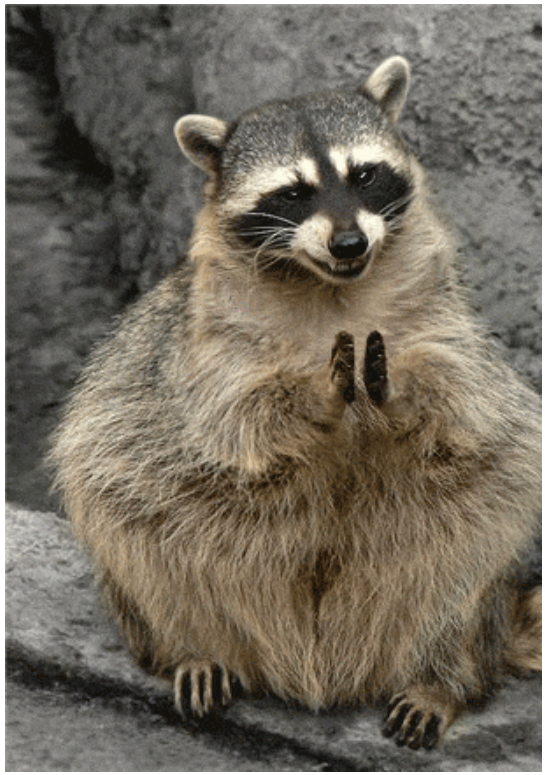


A veces desea aplicar una sola función muchas veces a través de las filas o columnas de un conjunto de datos

..Tambien puedes usar for… pero algunas veces es confuso

Las funciones apply aplican funciones a través de conjuntos de datos!

- `apply`
- `vapply`
- `lapply`
- `sapply`
- `tapply`
- `mclapply`



The `apply()` function

```
apply(X, MARGIN, FUN, ...)
```

- `X` = matriz
- `MARGIN`: 1 para aplicar a filas o 2 aplicar a columnas
- `FUN`: una función R (puede ser definida por el usuario)
- `...`: argumentos adicionales a la función nombrada en `FUN`

```
m <- matrix(1:12,nrow=3)
```

```
apply(X=m, MARGIN=2, FUN=mean)
[1] 2 5 8 11
```

```
apply(X=m, MARGIN=1, FUN=mean)
[1] 5.5 6.5 7.5
```

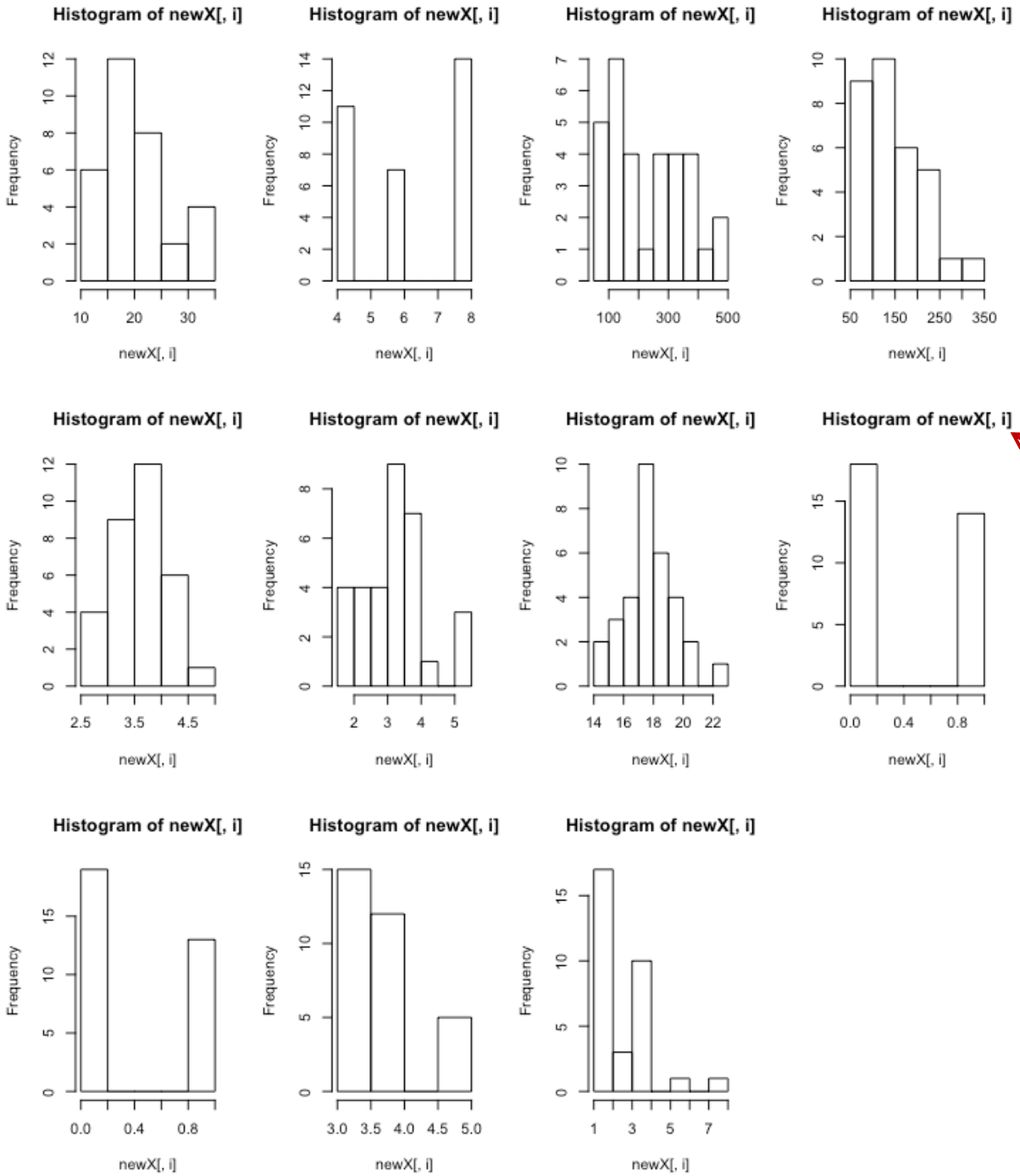
```
m
     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12
```

apply()

En realidad sólo está ejecutando un bucle for, pero el código es más fácil de escribir

```
par(mfrow=c(3,4))
apply(X=mtcars, MARGIN=2, FUN=hist)
```

Error in plot.new() : figure margins too large


Histogram of newX[, i]
Frequency
newX[, i]
Hey there is the for loop counter!

Las funciones apply aplican funciones a través de conjuntos de datos!

- `apply`
- **`vapply`**
- **`lapply`**
- **`sapply`**
- **`tapply`**
- **`mclapply`**

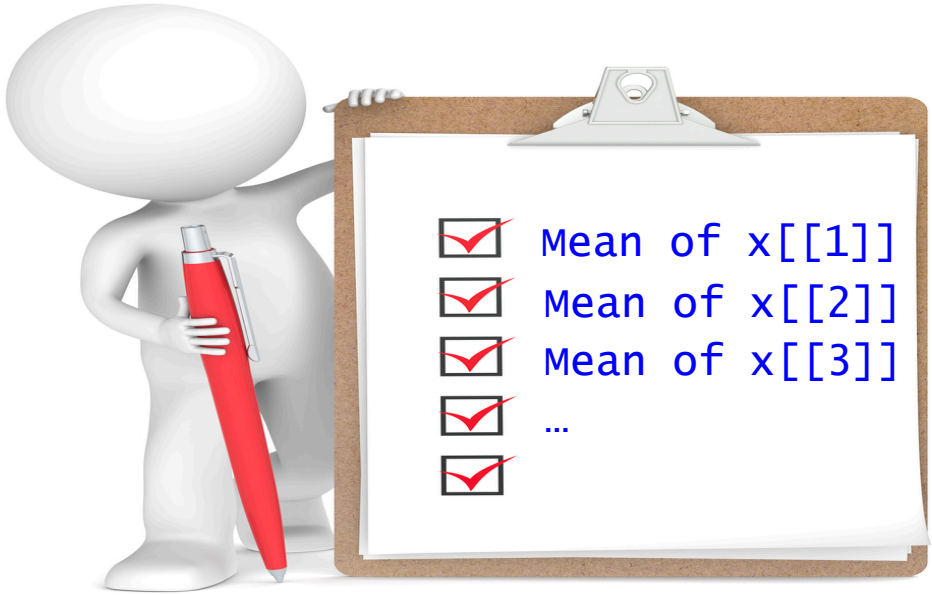
lapply()

- Aplica una función a los elementos de una **lista (list)**

- Devuelve una lista de la misma longitud que X.

```
x <- list(a = 1:10, b = 1:50)

# calcular la media para cada elemento de la lista
 lapply(x, mean)
```

```
$a
[1] 5.5

$b
[1] 25.5
```



tapply()

```
head(ToothGrowth)
  len supp dose
4.2   VC  0.5
11.5  VC  0.5
7.3   VC  0.5
5.8   VC  0.5
6.4   VC  0.5
10.0  VC  0.5
```

- Aplica una función a un vector, pero divide ese vector por alguna otra variable categórica
-

```
tapply(X, FUN, INDEX ...)
```

```
# Calcular el promedio de crecimiento de los dientes para cada tipo de suplemento de Guinea Pigs

tapply(X = ToothGrowth$len, FUN=mean, INDEX=ToothGrowth$sup)

OJ       VC
20.66333 16.96333
```

Las funciones apply aplican funciones a través de conjuntos de datos!

- `apply`
- `vapply`
- `lapply`
- `sapply`
- `tapply`
- `mclapply`

Explora las otras funciones apply con la ayuda de R!

?sapply

Ya casi acabamos con los fundamentos de programación III

- **Acelerar el código**
- Buenas prácticas de codigo
- Buscar ayuda!

Velocidad

La ejecución en la memoria del computador es 216 veces más rápido que imprimir los resultados en la consola > 54 veces más rápido

que escribir en un archivo

```
for (i in 1:1000000) { #0.25 seconds
   x <- i
}

for (i in 1:1000000) { #54.0 seconds
    print(i)
}
```

(NO HAGAS ESTO!)

```
for (i in 1:1000000) { #47 minutes
   write.csv(file="temp.csv", i)
}
```

Solución!

- Para un mayor velocidad `print()` o `write()` fuera de bucles

```
x <- vector(length=1000000)  #11.1 seconds for all
for (i in 1:1000000) {
   x[i] <- i
}

write.csv(file="temp.csv", x)
```

Ya casi acabamos con los fundamentos de programación III

- Acelerar el código
- **Buenas prácticas de código**
- Buscar ayuda!

Nunca son demasiados comentarios!!!

Verifique su código!

- Pruebe su código, pruebe su código, pruebe su código!
- Pruebe ejemplos sencillos a los que conozca la respuesta para asegurarse de que está recibiendo la respuesta que debe obtener!
- Grafique los resultados: es esto lo que esperabas?

Organice el código en scripts

- Dividir código muy largo en varios scripts

- Darles nombres informativos, y numerarlos si tienen un orden

mal=	**bien=**
algo.r	0_datos_organizados.R
codigo.r	1_analisis_exploratorio.R
	2_ analisis_PCA.R
	3_figuras_finales.R

Dentro de cada script, organice el código

- Cargar todos los paquetes en la parte superior
- Cargue los datos que necesita
- Usar espacios en blanco en el código: agregue líneas en blanco entre secciones de código

No tengas miedo de pedir ayuda

```
?function_name
```
o
```
?? function_name
```

Googlealo...

Lo mas importante
No te rindas!

R es un idioma, y será más fácil (y más divertido) cuanto más lo hagas.