Taller RLadies: R como lenguaje de programación

Algunas cositas sobre R markdown

Les cuento que este archivo lo generé con R Markdown!. Esta es mi primera experiencia con R Markdown, así que aprenderemos juntas (sobre todo en un próximo taller de RLadies sobre Markdown!). Les dejo los siguientes comandos que me tiró este template cuando lo abrí, asi ya nos vamos amigando.

- Try executing this chunk by clicking the *Run* button within the chunk or by placing your cursor inside it and pressing *Ctrl+Shift+Enter*.
- Add a new chunk by clicking the *Insert Chunk* button on the toolbar or by pressing *Ctrl+Alt+I*.
- When you save the notebook, an HTML file containing the code and output will be saved alongside it (click the *Preview* button or press *Ctrl+Shift+K* to preview the HTML file).
- The preview shows you a rendered HTML copy of the contents of the editor. Consequently, unlike *Knit*, *Preview* does not run any R code chunks. Instead, the output of the chunk when it was last run in the editor is displayed.
- Agrego el "Clásico comando" Ctrl+s para ir guardando el documento a medida que escribo. Igual cada vez que compilamos se guarda!
- Y también agrego el comando Ctrl+z , que es el "UNDO", y que me vuelve para atrás un paso. Por ejemplo cuando borro algo y me di cuenta de que me equivoqué.

Ahora si vamos al taller de hoy

En este taller veremos los siguientes conceptos de programación básica en R:

1. Qué son los loops? por qué son útiles?.
2. Loops con la sentencia *for*. Ejemplos.
3. Operadores lógicos
4. Mas loops, con la sentencia *while*.
5. Sentencias *if*, *else*. Ejemplos.
6. Tipos de variables para guardar información. Variables temporales.
    - PRACTICA: Ejercicios 1 a 3.
    - Ejemplo de calcular la riqueza (matriz de nxm)
    - PRACTICA: Ejercicio 4.
7. Funciones, que son para qué sirven? Como se escriben en R?. Ejemplos.
    - PRACTICA: Ejercicios 5 y 6.
8. Debbuging con el comando browser().

1. Loops o Bucles

Los loops nos sirven para repetir muchas veces una cantidad dada de instrucciones que se especifican entre **llaves** {}.

Sentencia *for*.

Con la sentencia *for*, la cantidad de instrucciones a repetir se indica con un índice que se escribe luego de la instrucción "for" entre **paréntesis** () (lo llamaremos "i" pero puede tener cualquier nombre). Ese índice puede o no usarse dentro del loop como variable.

Veamos algunos Ejemplos:

1. Imprimo los números del 1 al 5. Aquí uso el índice como variable a imprimir dentro del loop.

```
for(i in 1:5)
{
  print(i)
}
```

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

2. Imprimo 5 veces el número 1. Aqui NO uso el índice como variable a imprimir dentro del loop.

```
for(i in 1:5)
{
  print(1)
}
```

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

3. Los loops sirven también para llenar un vector con números:

```
#antes del loop tengo que definir los vectores vacíos que luego llenaré dentro del loop
vec1<-vector()
vec2<-vector()

#en el siguiente loop lleno un vector de dos maneras distintas
for(i in 1:10)
{
  #asignando elemento por elemento
  vec1[i]=i+1
  #usando c (concatenate o concatenar)
  vec2<-c(vec2,i+1)
}
```

Imprimo los vectores en pantalla

```
vec1
```

```
##  [1]  2  3  4  5  6  7  8  9 10 11
```

```
vec2
```

```
##  [1]  2  3  4  5  6  7  8  9 10 11
```

4. También con un loop puedo llenar una matriz con números:

```
#defino una matriz con todos sus elementos iguales a cero (si no especifico data
#la llena con NA)
unamatriz<-matrix(data=0,nrow=5,ncol=3)
#lleno la matriz usando dos loops uno que recorre todas las filas (indicefila)
#y otro que recorre todas las columnas (indicecolumna).
for(indicefila in 1:nrow(unamatriz))
```

```
  {
    for(indicecolumna in 1:ncol(unamatriz))
    {
      unamatriz[indicefila,indicecolumna]=8
    }
}
#imprimo en pantalla la matriz
unamatriz
```

```
##      [,1] [,2] [,3]
## [1,]    8    8    8
## [2,]    8    8    8
## [3,]    8    8    8
## [4,]    8    8    8
## [5,]    8    8    8
```

Aunque sabemos que esto mismo lo podría haber hecho en forma mas compacta haciendo:

```
unamatriz2<-matrix(data=8,nrow=5,ncol=3)
unamatriz2
```

```
##      [,1] [,2] [,3]
## [1,]    8    8    8
## [2,]    8    8    8
## [3,]    8    8    8
## [4,]    8    8    8
## [5,]    8    8    8
```

No siempre el hecho de usar loops es mejor. Muchas veces existen funciones que ya están implementadas en R y que son mas eficientes. Estas funciones prefabricadas en R-base no son otra cosa que loops, pero programados en algún lenguaje de mas bajo nivel (en general C) que hacen que la función sea más rápida.

5. Pero cuando queremos llenar una matriz con números que cambian con los índices de la matriz, los loops si son útiles. Por ejemplo llenemos una matriz con números que sean la suma de la fila y la columna correspondiente:

```
otramatriz<-matrix(data=0,nrow=4,ncol=2)
for(indicefila in 1:nrow(otramatriz))
  {
#indicecolumna recorre las columnas
    for(indicecolumna in 1:ncol(otramatriz))
    {
      otramatriz[indicefila,indicecolumna]=indicefila+indicecolumna
    }
}
otramatriz
```

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

Con la sentencia *for* el índice va recorriendo en forma secuencial una serie de valores que especificamos como vector luego de la palabra "in". Hasta ahora lo hicimos con índices consecutivos pero pueden no serlo, veamos un ejemplo:

```r
vectorpares<-seq(from=2,to=10,by=2) #vector de numero pares
vectorimpares<-vector()
for (ind in vectorpares)
{
  vectorimpares<-c(vectorimpares,ind+1)
}
vectorimpares
```

```
## [1]  3  5  7  9 11
```

También los índices pueden ser palabras:

```r
frutas<-c("banana","pera","manzana")
for (fr in frutas)
  {
  print(fr)
  }
```

```
## [1] "banana"
## [1] "pera"
## [1] "manzana"
```

3. Operadores lógicos

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

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

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

Veamos algunos ejemplos:

```r
10 == 10
```

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

```r
x <- 10
x == 10
```

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

```r
y <- NA
is.na(y)
```

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

```r
## podemos combinar expresiones condicionales con || y &&
is.na(y) && x==10
```

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

También son útiles las siguientes 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
```

4. Sentencia *while*

La forma general de un bucle con while es:

while (condición) { hacer algo }

mientras (*while*) la condición entre paréntesis sea verdadera, entonces se ejecutan las acciones dentro de las llaves. Vemos unos ejemplos:

1.

```
cuantos=1
while(cuantos<=10){
    cuantos=cuantos+1
    print(cuantos)
}
```

2.

```
numeroingresado<-readline("ingrese un número positivo ")
while(numeroingresado<=0){
    numeroingresado<-readline("ingrese un número positivo ")
    print("ingrese un número positivo")
}
print("Bien, ahora sigamos...")
```

5. Sentencia *if else is*

Con un *if* en un programa, preguntamos: **si** la expresión entre paréntesis es verdadera, entonces hacé todo lo que está entre llaves; sino salteá lo que está entre llaves y seguí con el programa.

if (expresión) { "hacer algo" }

En un *if*, la conversión de lo que está entre paréntesis a tipo boolean (VERDADERO/FALSO) es implícita. La sentencia completa if...else es:

if (expresión 1) {

"hacer una cosa si la expresión 1 es verdadera"

} else if (expresión 2) {

"hacer otra cosa si la expresión 1 es falsa y la expresión 2 es verdadera"

} else {

"hacer otra cosa distinta si la expresión 1 es falsa y expresión 2 es falsa"

}

Solo será hecha UNA de las cosas entre laves, dependiendo de las expresiones a evaluar.

Veamos algunos ejemplos: 1.

```
#asigno a x el valor 5
x<-5
if(x>3)
  {
    y<-10
```

```
  }else{
    y<-0
  }
#inprimo y en pantalla
y
```

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

Otra forma mas compacta es:

```
x<-5
y <- if(x > 3) { 10} else { 0}
y
```

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

6. Tipos de Variables

Para chequear de que tipo es un objeto se usa el comando class().

1. Números ("numeric")

```
a<-5
class(a)
```

```
## [1] "numeric"
```

```
b<-"5"
class(b)
```

```
## [1] "character"
```

```
c<-as.numeric(b)
class(c)
```

```
## [1] "numeric"
```

2. Nombres, letras o strings ("character")

```
name<-c("Clotilde")
class(name)
```

```
## [1] "character"
```

3. Vectores ("vector")

```
mivectorlogico<-vector(mode="logical",length=10)
mivectornumerico<-vector(mode="numeric")
for(i in 1:3)
 {
 mivectornumerico[i]=2*i
 }
mivectornumerico
```

```
## [1] 2 4 6
```

4. Matrices ("matrix")

```
mimatriznumerica<-matrix(data=0,nrow=5,ncol=5)
for (i in 1:nrow(mimatriznumerica))
  {
    for(j in 1:ncol(mimatriznumerica))
    {
    mimatriznumerica[i,j]=(i-1)*ncol(mimatriznumerica)+j
    }
  }
mimatriznumerica
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    2    3    4    5
## [2,]    6    7    8    9   10
## [3,]   11   12   13   14   15
## [4,]   16   17   18   19   20
## [5,]   21   22   23   24   25
```

5. Data Frames ("data.frame")

Son mas generales que una matriz porque sus elementos pueden ser de diferentes tipos (no necesariamente números como en una matriz). Son un caso particular de lista. Un dataframe es una lista en la que el número de filas es igual al número de columnas.

Por ejemplo:

```
undataframe <- data.frame("Identificador" = 1:2, "Edad" = c(21,15), "Nombre" = c("Ana", "Clara"), string
undataframe
```

```
##   Identificador Edad Nombre
## 1             1   21    Ana
## 2             2   15  Clara
```

IMPORTANTE: muchas funciones de lectura de datos como read.table(), read.csv(), read.delim(), read.fwf(), cargan los datos como un dataframe.

6. Listas ("list")

```
milistavariada<-list()
milistavariada<-list(mimatriznumerica,mivectorlogico,mivectornumerico)
milistavariada
```

```
## [[1]]
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    2    3    4    5
## [2,]    6    7    8    9   10
## [3,]   11   12   13   14   15
## [4,]   16   17   18   19   20
## [5,]   21   22   23   24   25
##
## [[2]]
```

```
##  [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##
## [[3]]
## [1] 2 4 6
```

Accedo a los elementos de la list con doble corchete

```
milistavariada[[3]]
```

```
## [1] 2 4 6
```

7. Funciones

Las funciones sirven para empaquetar un conjunto de instrucciones, tomando como entrada una serie de argumentos. En R se escriben de la seguiente manera:

```
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 <- tranca_funcion(arg2) # genero alto_resultado

return(alto_resultado)
}
```

En este pseudocódigo, mi función se llama "alta_funcion" (podría llamarse de otra manera) y toma como valores de entrada los valores que le ponga entre **paréntesis**(). Usando esos valores, la función realizará todo lo que le ponga entre **llaves**{} y devolverá como resultado solo lo que le ponga después del rerturn entre los **paréntesis**().

IMPORTANTE: si no le especifico return, devuelve la última línea de la función. Como buena práctica de programación, cuando una función retorna algo, conviene siempre indicar con return lo que la función devuelve.

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

Veamos un ejemplo:

```r
suma2vectores<-function(vector1=c(1,1),vector2=c(2,2))
{
vsuma<-vector()
for (i in 1:length(vector1))
  {
  vsuma[i]=vector1[i]+vector2[i]
  }
return(vsuma)
}
```

```r
suma2vectores()
```

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

```r
v1<-c(2,3,4)
v2<-c(1,1,1)
suma2vectores(v1,v2)
```

```
## [1] 3 4 5
```

Lo mismo lo hago con suma vectorizada, que es una función que ya viene con el R-base y que está optimizada (en general programada en C):

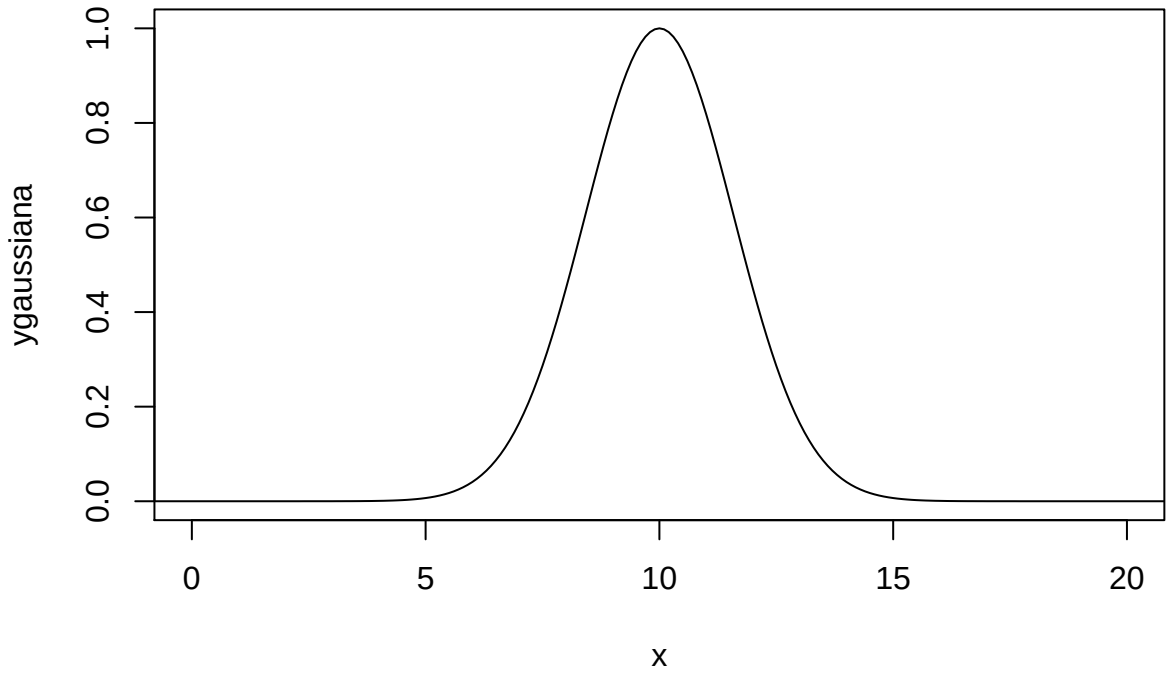
```r
v1+v2
```

```
## [1] 3 4 5
```

Otro Ejemplo:

Queremos evaluar una distribución de probabilidad, por ejemplo una gaussiana, que se define como:

```r
#gaussiana de media 10
media=10
#varianza 5
varianza=5
#altura max 1
amplitud=1
#evalúo en un vector x
x<-seq(from=-100,to=100,by=0.1)
ygaussiana=amplitud*exp(-(x-media)**2/(varianza))
#dibujo x,ygaussiana
plot(x,ygaussiana,xlim=c(0,20),ty="l")
```

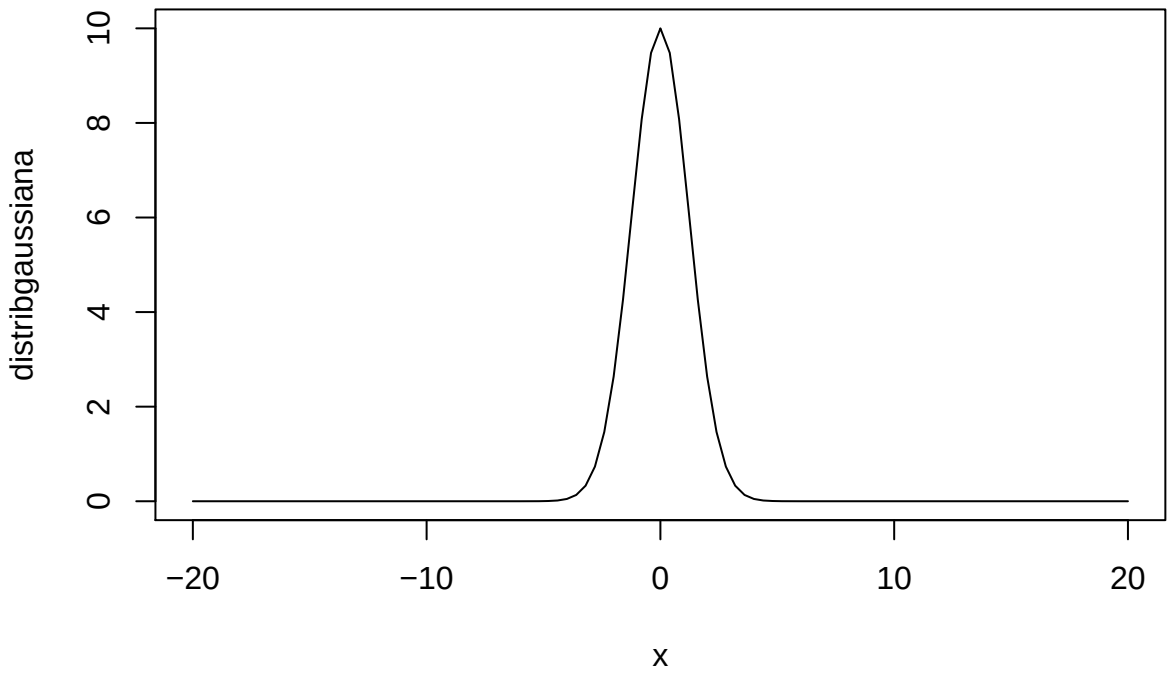


Ahora quiero cambiar facilmente la media, la varianza etc sin tener que cambiar sus valores en el código cada vez que quiero evaluar la función. Entonces me conviene crear una función, que tome como entrada los valores que voy a cambiar seguido (serán lo *ARGUMENTOS* de la función).

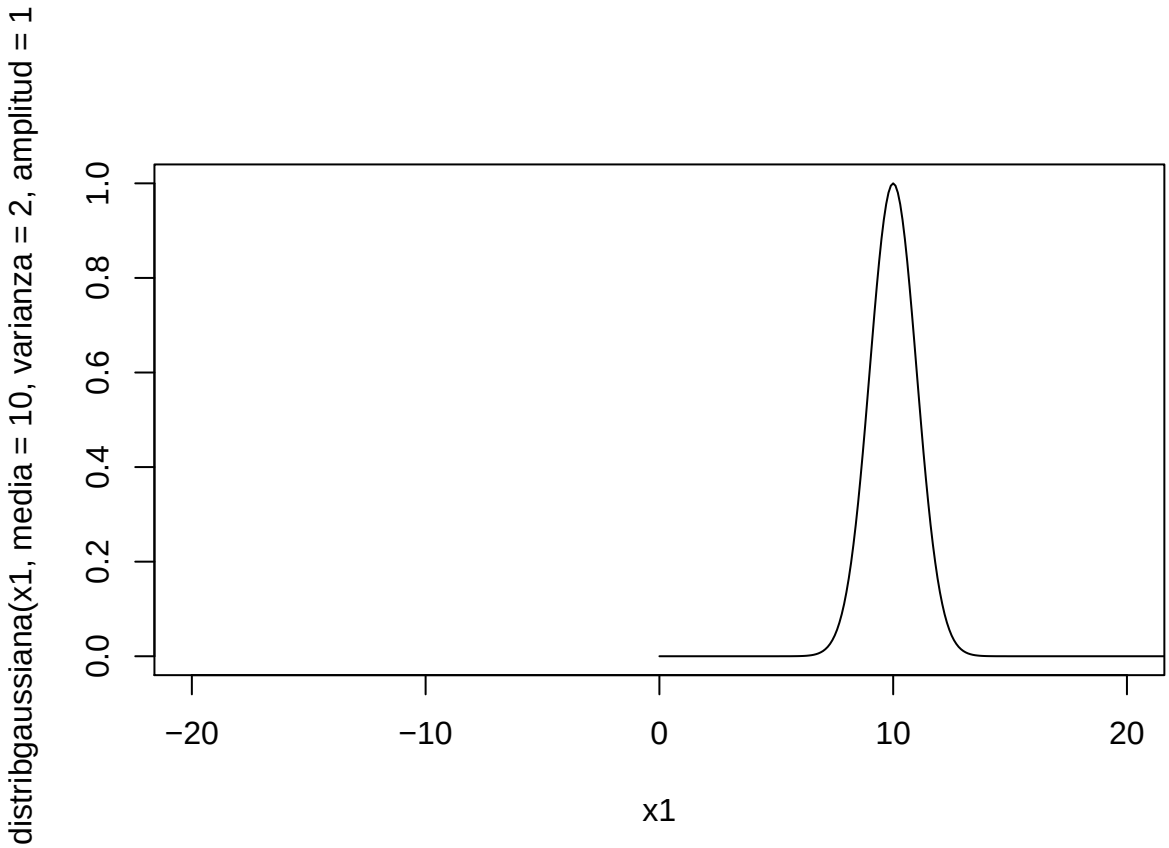
IMPORTANTE: Cuando inventamos un nombre para una función, no usar nombres de funciones ya definidas en R, por ejemplo: mean, sum, sd, etc. Además poner un nombre autoexplicativo de la función en cuestión. Por ejemplo: a la función sqrt la podría llamar raizcuad, o raiz2.

```
distribgaussiana<-function(x=seq(from=-100,to=100,by=1),media=0,varianza=3,amplitud=10)
{
gauss=amplitud*exp(-(x-media)**2/(varianza))
return (gauss)
}
```

```
plot(distribgaussiana,xlim=c(-20,20))
```




```
x1<-seq(0,100,0.1)
plot(x1,distribgaussiana(x1,media=10,varianza=2,amplitud=1),xlim=c(-20,20),ty="l")
```



No es necesario escribir los nombres de los argumentos de la función, pero suele hacerse para que el código sea más claro, sobre todo cuando lo compartimos o cuando lo volvemos a usar luego de un tiempo.

8. El comando browser()

Cuando no usamos R Studio para programar, el comando **browser()** es MUY IMPORTANTE y nos sirve para conocer los valores que van tomando las variables en cualquier punto del programa. El programa interrumpe su ejecución cuando lee este comando y me devuelve el prompt donde puedo preguntar cual es el valor de las variables. Por ejemplo escribiendo **list()** me lista todas las variables que el programa tiene en memoria. Escribiendo alguna de ellas me dará su valor.