Introducción a la Programación en R

Mihai M. Craiu

7/9/2020

Contents

Condicionales **1**

Ejercicios 4

Bucles **6**

Ejecicios 12

Funciones I **17**

Ejercicios 24

Condicionales

Las **estructuras selectivas** o **condicionales** son aquellas que nos permten hacer una seleccion entre dos o varias rutas de ejecución posible. La selección se llevará a cabo según el valor de una condicioón. en lo sucesivo entenderemos por expresion a cualquier agrupación de comandos entre llaves, bien separadas por punto y coma, o bien por salto de línea. La primera estructura condcional que vamos a conocer es **if**, y no es más que un **si condicional:**

if (condición1) expresión1
Si la condicion de **condición1** se verifica (TRUE), entonces la sentencia a ejecutar recogida en *expresión1* se lleva a cabo. Si el resultado de la evaluación es falso, la sentencia a ejecutar simplemente se ignora.

Ejemplo: Número positivo

```
x <- 20
if (x>0) {paste(x,"es positivo")}
```

```
## [1] "20 es positivo"
```

Ejemplo: Número positivo, negativo o cero

```
y <- -15
if (y>0) {paste(y, "es positivo")}
if (y==0) {paste(y,"es cero")}
if (y<0) {paste(y, "es negativo")}
```

```
## [1] "-15 es negativo"
```

Ejemplo: Número par o impar (supone un uso interactivo de R)

```
n <- readline(prompt = "Introduzca un entero: ")
n <- as.numeric(n)
if (n%%2 == 0) {paste(n, "es par")}
if (n%%2 == 1) {paste(n, "es impar")}
```

En los dos últimos ejemplos hemos visto que a veces necesitamos que una sentencia se ejecute cuando se cumple la condición, y que otra sentencia se ejecute cuando dicha condición sea falsa. Si nos encontramos en este caso, la forma de resolverlo únicamente a través de sentencias **if** es engorrosa y poco práctica, por lo que existe otra forma más sencilla para llevar a cabo dicha tarea: la estructura **if-else:**:

if (condición1) expresión1 else expresión2
Si **condición1** es evaluada com verdadera, entonces se ejecuta la **expresión1**, en cambio, cuando la c ondición se evalúa como falsa, se ejecuta la **expresión2**

Las estructuras if-else se pueden **anidar** y el lenguaje R dispone además de una version vectorizada de if, *ifelse()*, sobre la que volveremos posteriormente.

Ejemplo: Número par o impar (supone un uso interactivo de R)

```
n <- readline(prompt= "Introduzca un entero: ") n <- as.numeric(n) if (n%%2 == 0) {paste(n,
"es par")} else {n, "es impar"}
```

```
y <- -15
if (y>0){
    paste(y, "es positivo")
}else if (y==0){
    paste(y, "es cero")
}else {paste(y, "es negativo")}
```

```
## [1] "-15 es negativo"
```

Ejemplo: Tipo de Bono

```
Sector_Emisor <- "Industria_Textil"
Vencimiento <- 5

if ( (Sector_Emisor=="Gobierno") | (Sector_Emisor=="GOBIERNO")){
    if(Vencimiento<=1){
        Clasif_Bono <- "GOB_CORTO"
    }else if (Vencimiento<=3){
        Clasif_Bono <- "GOB_MEDIO"
    }else{Clasif_Bono <- "GOB_LARGO"}
}else if ((Sector_Emisor=="Financiero") | (Sector_Emisor=="FINANCIERO")){
```

```
        if(Vencimiento<=1){
            Clasif_Bono <- "FIN_CORTO"
        }else if(Vencimiento>=3){
            Clasif_Bono <- "FIN_MEDIO"
        }else{Clasif_Bono <- "FIN_LARGO"}
}else {
        if(Vencimiento<=1){
            Clasif_Bono <- "NOFIN_CORTO"
        }else if (Vencimiento<=3){
            Clasif_Bono <- "NOFIN_MEDIO"
        }else{Clasif_Bono <- "NOFIN_LARGO"}
}
Clasif_Bono
```

```
## [1] "NOFIN_LARGO"
```

Ejemplo: Múltiplos de 4

```
m <- 15
cociente <- floor(m/4)
if(m%%4 ==0){
    paste(m,"es de la forma 4k, con k=", cociente)
}else if(m%%4==1){
    paste(m,"es de la fomra 4k+1, con k=", cociente)
}else if(m%%4==2){
    paste(m,"es de la forma 4k+2, con k=", cociente)
}else {paste(m,"es de la forma 4k+3,con k=", cociente)}
```

```
## [1] "15 es de la forma 4k+3,con k= 3"
```

A menudo sucede, como en el ejemplo anterior, que tenemos difernetes opciones dependiendo del valor de una variable. Para evitar la concatenación de excesivos condicionales if, tenemos a nuestra disposición la estructura **switch**:

switch(varaible, expresión1, expresión2, . . . ,expresionN)
Si **varaible** es un entero positivo i, entonces se ejecuta la expresión i-ésima.

switch(varaible, string1=expresión1, . . . , stringN=expresiónN)
Si **variable** es del tipo cadena de caracteres, entonces se ejecuta la expresión cuyo identificador coincida con el contenido de la variable. A diferencia del caso numérico, en caso de no coincidencia de la variable con los indetificadores, se ejectura la expresión por defecto **default**.

Ejemplos:

```
switch(1,"Madrid","Barcelona","Valencia")
```

```
## [1] "Madrid"
```

```r
switch(3,"Madrid","Barcelona","Valencia")
```

```
## [1] "Valencia"
```

```r
switch(0,"Madrid","Barcelona","Valencia")
switch("tres",uno="1",dos="2",tres="3")
```

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

```r
switch("siete",uno="1",dos="2",tres="3","Otro")
```

```
## [1] "Otro"
```

Ejemplo: Tipo de seguro

```r
Identif_seguro <- 3
switch(Identif_seguro,
        {TIPO_SEGURO <- "Hogar"},
        {TIPO_SEGURO <- "Autos"},
        {TIPO_SEGURO <- "Vida"})
TIPO_SEGURO
```

```
## [1] "Vida"
```

Ejemplo: Múltiplos de 4... ¿es correcto?

```r
m <- 9
cociente <- floor(m/4)
resto <- m%%4
switch(resto,
        {paste(m,"es de la forma 4k+1, con k =", cociente)},
        {paste(m,"es de la forma 4k+2, con k =", cociente)},
        {paste(m,"es de la forma 4k+3, con k =", cociente)},
        paste(m,"es de la forma 4k, con k =", cociente) )
```

```
## [1] "9 es de la forma 4k+1, con k = 2"
```

Ejemplo: Múltiplos de 4

```r
m <- 12
cociente <- floor(m/4)
resto <- as.character(m%%4)
switch(resto,
        "1" = {paste(m,"es de la forma 4k+1, con k =", cociente)},
        "2" = {paste(m,"es de la forma 4k+2, con k =", cociente)},
        "3" = {paste(m,"es de la forma 4k+3, con k =", cociente)},
        paste(m,"es de la forma 4k, con k =", cociente) )
```

```
## [1] "12 es de la forma 4k, con k = 3"
```

Ejercicios

Ejercicio propuesto: Clasificación por edad

Escribe un código que permita clasificar la variable RIESGO_ASEGURADO en "Alto", "Bajo" o "Moderado", en función de la variable EDAD_ASEGURADO, según que ésta sea menor de 25 años o mayor que 70.

```
EDAD_ASEGURADO <- 35

if (EDAD_ASEGURADO < 25){
    RIESGO_ASEGURADO <- "Alto"; RIESGO_ASEGURADO
} else if (EDAD_ASEGURADO > 70) {
    RIESGO_ASEGURADO <- "Medio"; RIESGO_ASEGURADO
} else {RIESGO_ASEGURADO <- "Bajo"; RIESGO_ASEGURADO}
```

```
## [1] "Bajo"
```

Ejercicio propuesto: Clasificación por edad y sexo

Modifica el código anterior para que, en caso de que la variable SEXO ASEGURADO sea "M"(mujer, frente a "H", hombre), el correspondiente perfil de riesgo disminuya un nivel.

```
EDAD_ASEGURADO <- 18
SEXO_ASEGURADO <- "M"

if (EDAD_ASEGURADO < 25){
    RIESGO_ASEGURADO <- "Alto"
    if (SEXO_ASEGURADO == "M") {RIESGO_ASEGURADO <- "Medio"}
    RIESGO_ASEGURADO
} else if (EDAD_ASEGURADO > 70) {
    RIESGO_ASEGURADO <- "Medio"
    if (SEXO_ASEGURADO == "M") {RIESGO_ASEGURADO <- "Bajo"}
    RIESGO_ASEGURADO
} else {RIESGO_ASEGURADO <- "Bajo"; RIESGO_ASEGURADO}
```

```
## [1] "Medio"
```

Ejercicio propuesto: Opción call

Escribe un código que permita obtener el benefcio/pérdida para un inversor que compra por 1e el díaa 3 de septiembre de 2018 una opción de compra (call) sobre unas acciones de Telefónica con vencimiento el 28 de septiembre de 2018, a un precio de ejercicio de 7.19e.

```
Strike <- 7.19
Prima <- 1
PyG <- 0

Cotiz_Telef_28_09_2018 <- 8.19

if(Cotiz_Telef_28_09_2018 <= Strike){
```

```
        PyG <- -Prima; PyG
} else {PyG <- Cotiz_Telef_28_09_2018 - (Strike + Prima); PyG}
```

```
## [1] -1.776357e-15
```

```
#Observación: Para R, debido a la aritmética de punto flotante que emplea para representar
#números reales, sucede que
sqrt(2)^2 == 2              #Arroja FALSE
```

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

```
#En tal caso es preferible hacer:
all.equal(sqrt(2)^2, 2)     #Devuelve TRUE
```

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

```
#En consecuencia,
all.equal(0, PyG)
```

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

Bucles

Los **bucles** o **ciclos** son estructuras que nos permiten repetir la misma instrucción un número determinado de veces, o hasta que se verifique cierta condición. Como veremos posteriormente, el carácter vectorial de R permite prescindir a menudo de este tipo de estructuras (que además pueden llegar a relantizar mucho la ejecución), si bien hay situaciones en que es imprescindible hacer uso de las mismas, además de ser uno de los pilares báscios de la programación. El primer bucle que veremos es el bucle **for**:

for(variable1 in vector1) expresión1
Se evalúa la **expresión1** para cada uno de los valroes que **variable1** toma en el **vector1**.

Ejemplos:

```
for(i in 1:10) {print(paste("Caso",toString(i)))}
```

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

```r
for(j in c("Madrid","Valencia","Barcelona")) {print(j)}
```

```
## [1] "Madrid"
## [1] "Valencia"
## [1] "Barcelona"
```

```r
suma <-0; for(i in 1:100) {suma <- suma + i}; suma
```

```
## [1] 5050
```

```r
temp <-2; for(k in c(1,3,5,7,9)) {temp<- temp + k}; temp
```

```
## [1] 27
```

Ejemplo: Factorial de un número

```r
N <- 5
factorial_N <- 1
for(i in 1:N){factorial_N<- factorial_N*i}
factorial_N
```

```
## [1] 120
```

Ejemplo: Lista de pares e impares

```r
for(i in 1:14){
    if(i%%2==0){print(paste(toString(i),"es par"))}
    else{print(paste(toString(i),"es impar"))}
}
```

```
## [1] "1 es impar"
## [1] "2 es par"
## [1] "3 es impar"
## [1] "4 es par"
## [1] "5 es impar"
## [1] "6 es par"
## [1] "7 es impar"
## [1] "8 es par"
## [1] "9 es impar"
## [1] "10 es par"
## [1] "11 es impar"
## [1] "12 es par"
## [1] "13 es impar"
## [1] "14 es par"
```

Como hemos visto, el bucle *for* es muy útil para repetir algo cuando conocemos el número de repeticiones previamente. Sin emabrgo, en algunas ocasiones no es posible conocer dicho número de antemano, sino que necesitamoes un ciclo que pueda ser controaldo por una condición, y la evaluación de esta condición dependerá del estado del programa en un momento dado. El bucle **while** nos permite hacer esto:

while (condición1) expresión1
Se evalúa la **expresión1** mientras *condición1** es cierta

Los bucles *while*, aunque muy útiles, se emplean en la práctica menos que los bucles *for*, dado que es posible modificar estos últimos para que se interruman cuando se presenta una condición dada, como veremos posteriormente.

Ejemplo: Contador

```
i<- 1
while(i <=10) {
    print(paste("Caso", toString(i)))
    i <- i+1
}
```

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

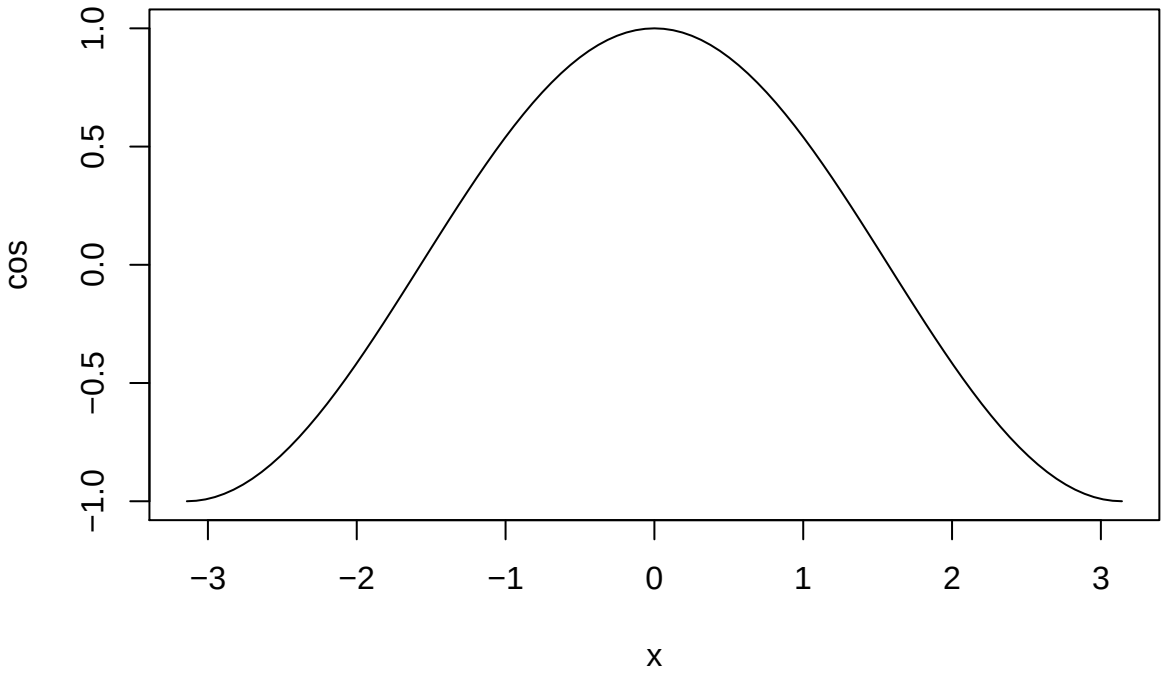
Ejemplo: Factorial de un número

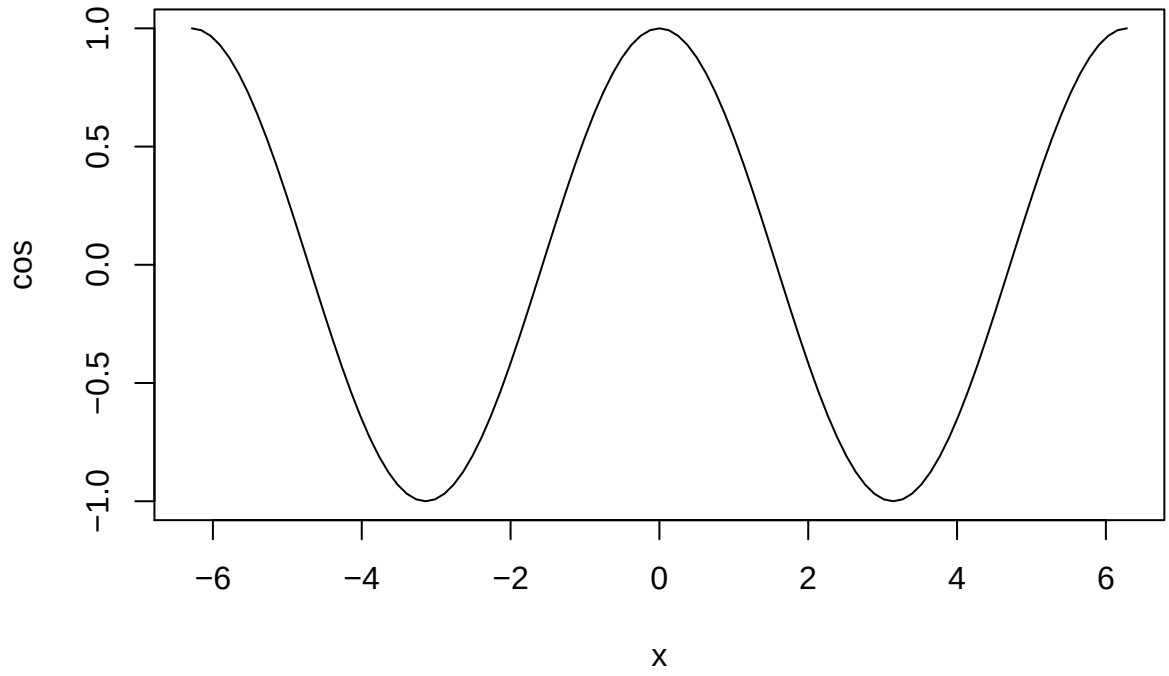
```
N <- 5
factorial_N <- N
while(N > 1){
    N <- N -1
    factorial_N <- factorial_N * N
}
factorial_N
```

```
## [1] 120
```

Ejemplo: Animación función coseno

```
for(i in 1:2){
    plot(cos, -i*pi, i*pi)
    #Vamos a almacenar en una variable la hora del sistema
    date_time <- Sys.time()
    #Con un bucle while vamos a controlar que el programa no haga nada durante 1 segundo
    while((Sys.time()-date_time)<1){}
}
```


cos
1.0
0.5
0.0
−0.5
−1.0
−3
−2
−1
0
1
2
3
x



Una estructura similar al ciclo *while*, en cuanto a que necesita evaluar una condición de parada para concluir las iteraciones, es el bucle **repeat**:

repeat expresión1
Se evalúa iterativamente la **expresión1**. Demanda una sentencia **break** para poder concluir.

Como se ha comentado, para poder concluir un bucle *repeat* es necesaria una sentencia **break**. Dicha instrucción también finaliza cualquier bucle *for* o *while*, simpre y cuando se verifique la conición que la acompaña. Por otro lado, el comando **next** dentro de un bucle también genera una interrupción del mismo, ahora bien, no de todo el bucle, sino de la iteración en que éste se encuentra.

Ejemplo: Contador

```
i <- 1
repeat {
    print(paste("Caso", toString(i)))
    i <- i+1
    if (i > 10) {break}
}
```

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

```
## [1] "Caso 6"
## [1] "Caso 7"
## [1] "Caso 8"
## [1] "Caso 9"
## [1] "Caso 10"
```

Ejemplo: Parchís

```
dado <- 1:6; n_tirada <- 0
repeat {
    n_tirada <- n_tirada + 1
    resultado_tirada <- sample(dado, size = 1)
    print(paste("En la tirada", toString(n_tirada),
                "ha salido un", toString(resultado_tirada)))
    if (resultado_tirada == 5) {
        print("Comienzas a jugar!!")
        break
    }
}
```

```
## [1] "En la tirada 1 ha salido un 1"
## [1] "En la tirada 2 ha salido un 4"
## [1] "En la tirada 3 ha salido un 4"
## [1] "En la tirada 4 ha salido un 1"
## [1] "En la tirada 5 ha salido un 6"
## [1] "En la tirada 6 ha salido un 6"
## [1] "En la tirada 7 ha salido un 1"
## [1] "En la tirada 8 ha salido un 6"
## [1] "En la tirada 9 ha salido un 4"
## [1] "En la tirada 10 ha salido un 6"
## [1] "En la tirada 11 ha salido un 5"
## [1] "Comienzas a jugar!!"
```

Ejemplo: Uso de break en for y while

```
for (i in 1:100){
    print(i)
    if (i == 8) {break}
}
```

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

```
k <- 0
while(k <=100){
    k <- k+1
```

```
    print(k)
    if(k == 7) {break}
}
```

```
## [1] 1
## [1] 2
## [1] 3
## [1] 4
## [1] 5
## [1] 6
## [1] 7
```

Ejemplo: Uso de next

```
for(i in 1:20){
    if(i %% 3 == 0) {next}
    print(i)
}
```

```
## [1] 1
## [1] 2
## [1] 4
## [1] 5
## [1] 7
## [1] 8
## [1] 10
## [1] 11
## [1] 13
## [1] 14
## [1] 16
## [1] 17
## [1] 19
## [1] 20
```

Ejecicios

Ejercicio propuesto: Omisión de iteraciones

Escribe un bucle que imprima por pantalla los números del 1 al 20 omitiendo las iteraciones que van de la 11 a la 15.

```
for (i in 1:20){
    if ((i>=11)&(i<=15)) {next}
    print(i)
}
```

```
## [1] 1
## [1] 2
## [1] 3
```

```
## [1] 4
## [1] 5
## [1] 6
## [1] 7
## [1] 8
## [1] 9
## [1] 10
## [1] 16
## [1] 17
## [1] 18
## [1] 19
## [1] 20
```

```r
#ALTERNATIVA: Operador %in% (?match):

for (i in 1:20) {
  if (i %in% 11:15)
    next
  print(i)
}
```

```
## [1] 1
## [1] 2
## [1] 3
## [1] 4
## [1] 5
## [1] 6
## [1] 7
## [1] 8
## [1] 9
## [1] 10
## [1] 16
## [1] 17
## [1] 18
## [1] 19
## [1] 20
```

Ejercicio propuesto: Valor Actual Bono

Calcula el precio de mercado de un bono de nominal 100.000, que paga cupone de 1000 durante los próximos 10 años y cuyo TIR es del 3,12%.

```r
VA_bono <- 0
Cup <- 1e3
Nominal <- 1e5
i <- 0.0312
for (n in 1:10) {
    VA_bono <- VA_bono + Cup*(1+i)^(-n)
    if(n==10){
        VA_bono <- VA_bono + Nominal*(1+i)^(-n)
    }
}
```

```r
VA_bono
```

```
## [1] 82026.22
```

Ejercicio propuesto: Número primo

Escribe un código que permita determinar si un número es primo.

```r
n <- 91
n_primo <- TRUE

for(i in 2:(sqrt(n))){
    if (n%%i == 0) {
        n_primo <- FALSE
        if (n==2) {n_primo <- TRUE}
        break
    }
}
n_primo
```

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

Ejercicio propuesto: Movimiento Browniano Geométrico

Un activo de renta variable puede modelizarse por medio de un Movimeinto Browniano Geométrico, de tal manera que, en vritud del Lema de Itô, el valor del activo en el instante t viene dado por la expresión

$$S_t = S_{t-1} e^{(\mu - \frac{\sigma^2}{2})\Delta t + \sigma\zeta\sqrt{\Delta t}}$$

donde μ y σ son respectivamente la deriva y la volatilidad del activo y $\zeta \sim$ N(0,1). Determina un posible valor de la cotización de una acción al cabo de 12 mese, suponiendo que S_0 =25, μ = 15% anual, σ =20% anual y Δt =1 mes.

```r
P_0 <- 25
mu <- 0.15
sigma <- 0.20
incr <- 1/12

P_t <- P_0

for(i in 1:12){
        P_t <- P_t*exp( (mu-0.5*sigma^2)*incr + sigma*sqrt(incr)*rnorm(1) )
}
P_t
```

```
## [1] 25.23418
```

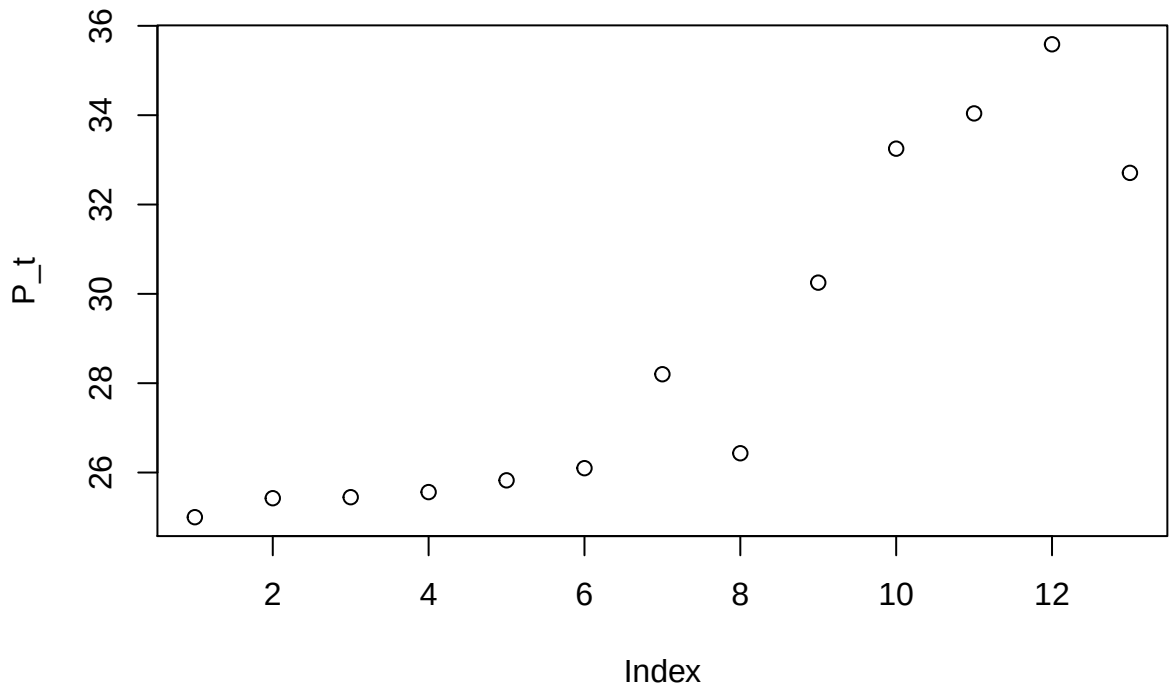
```r
#Senda: (con vectores)
P_0 <- 25
mu <- 0.15
sigma <- 0.20
incr <- 1/12

P_t <- c(P_0)

for(i in 1:12){
        P_t <- c(P_t,P_t[i]*exp( (mu-0.5*sigma^2)*incr + sigma*sqrt(incr)*rnorm(1) ))
}
P_t
```

```
##  [1] 25.00000 25.42507 25.44747 25.56183 25.82514 26.09732 28.20014 26.43140
##  [9] 30.25119 33.25037 34.03972 35.58695 32.70819
```

```r
length(P_t)
```

```
## [1] 13
```

```r
plot(P_t)
```



Ejercicio propuesto: Mímino de una función

Escribe un código que permita determinar de manera aproximada el mínimo de la funcio $f(x) = 6x^2 - 36x + 55$ en el intervalo [1 , 4].

```
N_subint <- 1e3
a <- 1
b <- 4
incr <- (b-a)/N_subint

xmin <- 1
ymin <- 6*xmin^2 - 36*xmin + 55

x <- 1
while (x <= 4){
        y <- 6*x^2 - 36*x + 55
        if (y < ymin ) {
            xmin <- x
            ymin <- y
        }
        x <- x +incr
}
print(paste("El mínimo se alcanza en x =",toString(xmin),", y =",toString(ymin)))
```

```
## [1] "El mínimo se alcanza en x = 3.001 , y = 1.000006"
```

A la hora de programar, una parte muy importante del trabajo recae en crear código que sea eficiente, en el sentido de que desempeñen la tarea propuesta de la manera más rápida posible. Es habitaul que en los códigos más eficientes la estructura del algoritmo implementado no sea explícita, mientras que los códigos simples en los que queda totalmente claro cómo se obtienen las cantidades a determinar, suelen ser bastante lentos computacionalmente. A continuación se muestra un ejemplo de cómo podemos medir con R el tiempo en el que se lleva a cabo un algoritmo:

Ejemplo: Medida del tiempo de ejecución

```
t_inicio <- proc.time()
for(i in 1:11){print(i)}
```

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

```
t_ejec <- proc.time()-t_inicio
t_ejec
```

```
##    user  system elapsed
##       0       0       0
```

Para concluir esta introdución a los bucles, comentar brevemente que en R, al igual que en otros lenguajes de programación, es posible realizar cálculos de una manera más rápida usando varios procesadores, todos ellos cooperando en una tarea determinada. Es lo como se conoce como **paralelizar** un proceso. La idea subyacente al procesamiento en paralelo es dividir la tarea entre múltiples procesadores y posteriormente volver a unir los distintos componentes. Si bien lo anterior se escapa totalmente de los objetivos del curso, se recomienda al lector interesado la lectura de la documentación del paquete de R **parallel**.

Funciones I

Ya hemos hecho uso de algunas funciones de R en los ejemplos anteriores. Una **función** es un elemento del programa que contiene código y se puede ejecutar, esto es, llevar a cabo una operación determinada. La funciónpuede **llamarse** o **invocarse** cuando sea necesario, de forma que el código que se enceuntra en su interior pasa a ejecutarse; una vez que la funcion termina, el programa continúa en la sentencia siguiente. La sintaxis de una función es la siguiente:

|***nombre <- funcion(arg_1, arg_2, ...) expresión1***| |:----------:-:-
|**expresión1** es una expresión de R que emplea los argumenteos, **arg_i**, para calcular un valor. El valor de la expresión1 es el valor que devuelve ka función (más concretamente, el último comando evaluado dentro de la expresión), y puede ser un numero, un vector, una lista, etc.| Una llamada a la función tiene habitualmente la forma nombre(arg_1, arg_2, ...)

Ejemplo: Función que multiplica dos números:

```
multiplicar_numeros <- function(x,y) {x*y}
multiplicar_numeros(2,7)
```

```
## [1] 14
```

Ejemplo: Factorial de un número

```
mi_factorial <- function(n){
    factorial<-1
    for (i in 1:n){
        factorial <- factorial *i
    }
    factorial
}
mi_factorial(5)
```

```
## [1] 120
```

La función se convierte en un objeto más de R, de modo que puede listar *lx()*, borrar *rm()*, etc. En algunos lenguajes de programación es obligatorio el uso del comando **return** para que la función deveulva un valor; sin embargo, en R no: como hemos comentado, una función de R devuelve el último valor calculado dentro de su cuerpo. Esto no impide que, en aras de una mayor claridad, podamos incluir dicho comando:

Ejemplo: Inclusión de comando return

```r
eleva_cuadrado1 <- function(x){x^2}
eleva_cuadrado1(7)
```

```
## [1] 49
```

```r
eleva_cuadrado2 <- function(x){return(x^2)}
eleva_cuadrado2(7)
```

```
## [1] 49
```

Al invocar una función los argumentos deben proporcionarse en el orden en el que se han definido doentro de la misma, a menos que éstos se den precedidos por su nombre. Elnombre de un argumento puede omitirse siempre que se pueda distinguir del resto de argumentos. El comando *args()* nos muestra los argumetnos de cualquier función, y más aún, podemos ver el contenido de cualquier función, y que hayamos creado tecleando simplemente su nombre.

Ejemplo: Función Exponencial

```r
exponencial <- function(base, exponente){base^exponente}
exponencial(2,3)
```

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

```r
exponencial(3,2)
```

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

```r
exponencial(base=2,exponente = 3)
```

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

```r
exponencial(exponente = 3,base = 2)
```

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

```r
exponencial(exponente = 3,2)
```

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

En muchas ocasiones interesa que la función que vamos a definir tenga argumentos con valores por defecto. De este modo, si por alguna razón dichos argumentos son omitidos en la llamada a la función, éstos tomarán el valor asignado en la definición de la misma. Los valores por defecto pueden ser cualquier expresión, no se restringen a constantes.

Ejemplo: Función exponencial con base e por efecto

```r
#Recordemos que e = exp(1)
exponencial <- function(base=exp(1),exponente){base^exponente}
exponencial(2,3)
```

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

```r
exponencial(3,2)
```

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

```r
exponencial(base = 2, exponente = 3)
```

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

```r
exponencial(exponente = 2)
```

```
## [1] 7.389056
```

```r
exponencial(2) #no es correcto??
```

```
## Error in exponencial(2): el argumento "exponente" está ausente, sin valor por omisión
```

```r
exponencial(,2)
```

```
## [1] 7.389056
```

Ejemplo: Valor por defecto no constante

```r
suma <- function(a=0,b=a){a+b}
suma(3,2); suma(5)
```

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

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

A veces puede ser interesante tener a nuestra disposición los argumentos de otras funciones. El argumento libre ... permite incluir en la definición de una función la posibilidad de invicarla incorporando los argumentos que puedan utilizar las funciones que formen parte del código de nuestra función.

Ejempo: Funcion logaritmo

```r
#Recordemos la funcion logaritmo en R:

log
```

```
## function (x, base = exp(1))  .Primitive("log")
```

```r
logaritmo1 <- function(x){log(x)}
logaritmo1(10)
```

```
## [1] 2.302585
```

```r
logaritmo1(100,base=10)
```

```
## Error in logaritmo1(100, base = 10): unused argument (base = 10)
```

Ejemplo: Función logaritmo con el argumento ...

```r
logaritmo2 <- function(x, ...){log(x,...)}
logaritmo2(10)
```

```
## [1] 2.302585
```

```r
logaritmo2(100,base=10)
```

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

Ejemplo: Función logaritmo con el argumento ... ¿es correcto?

```r
logaritmo3 <- function(x, ...){log(x)}
logaritmo3(10)
```

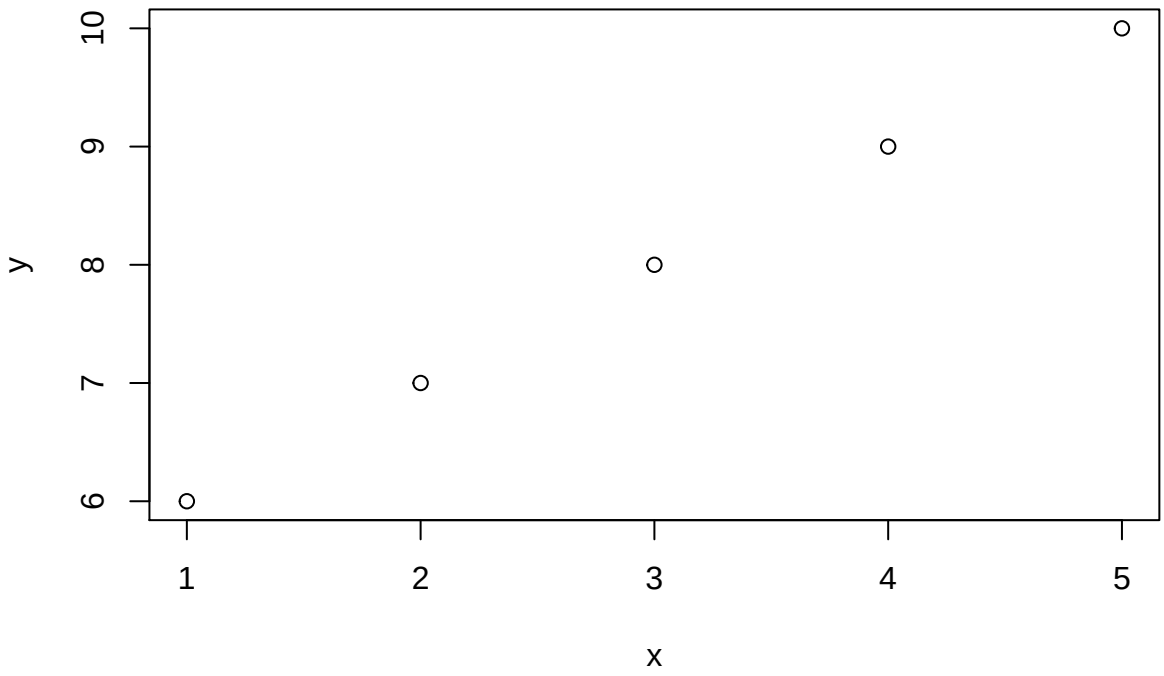
```
## [1] 2.302585
```

```r
logaritmo3(100,base=10)
```

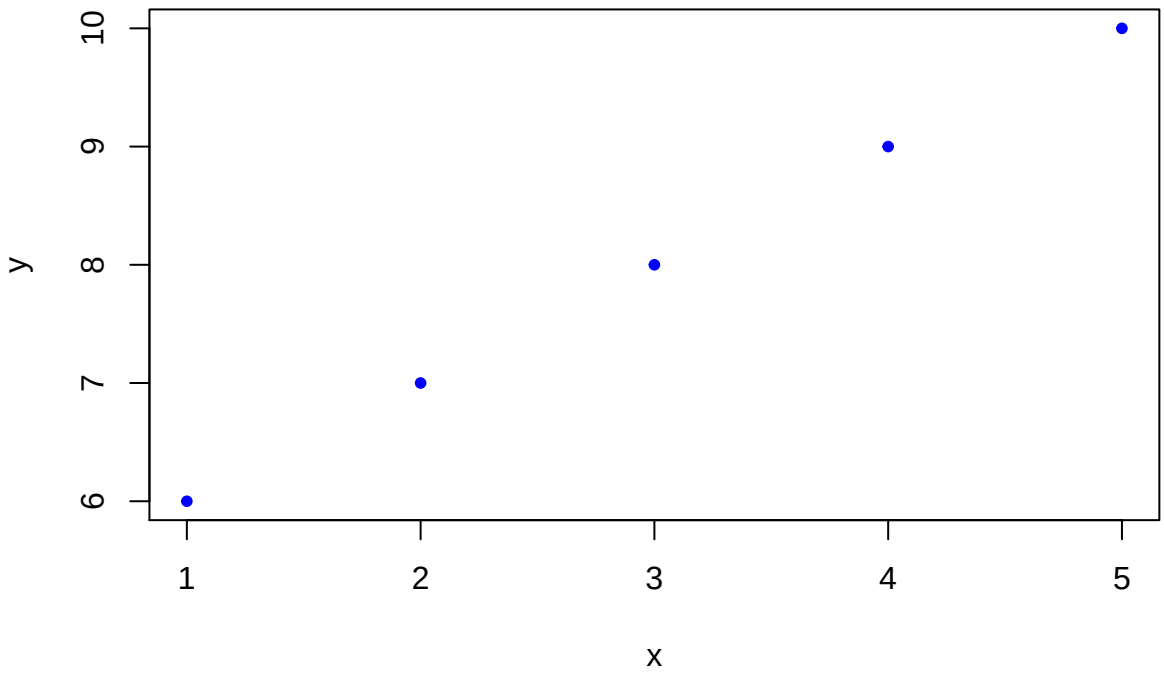
```
## [1] 4.60517
```

Ejemplo: Gráfica

```r
dibuja_func <- function(x, y, ...){plot(x, y, ...)}
dibuja_func(1:5, 6:10)
```




```
dibuja_func(1:5, 6:10, col= "blue", pch=20)
```



Culaquier asignación ordinaria realizada dentro de una función es local y temporal, y se pierde tras salir de la función.

Ejemplo

```
x <- 7
y <- 3
producto <- function(a,b){x <- a*b; x}
producto(x,y)
```

```
## [1] 21
```

```
x
```

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

Las variables que aparecen en el cuerpo de una función pueden ser de tres tipo:

- *párámetros formales*: son los argumentos de la función. Su valor queda determinado en el proceso de invocado de la función.
- *varaibles locales*: son aquellas cuyos valores quedan determinados al evaluar las distintas expresiones del ceurpo de la función.
- *variables libres*: son el resto de variables. Las varaibles libres pasan a ser locales si se asignan de algún modo en el cuerpo de la functión.

Ejemplo: Parámetros, varaibles locales y libres

```
prueba1 <- function(x) {
    y <- 2*x
    print(x)
    print(y)
    print(z)
}
```

En este ejemplo x es un parámetro formal, y es una varaible local y z es una varaible libre. Al invocar la función *prueba1*, R busca en primer lugar el valor de la variable z en el entorno que ha creado; como no la encuentra, pasa a buscarla en el exterior de la función. Finalmente, como sigue sin encontrarla, muestra un mensaje de error:

```
prueba1(5)
```

```
## [1] 5
## [1] 10

## Error in print(z): objeto 'z' no encontrado
```

Si ahora definimos la varaible z fuera del ámbito de la funcion *prueba1* y volvemos a invocarla, esta vez R sí es capaz de econtrar dicha variable y puede mostrar su valor:

```
z <- 1
prueba1(5)
```

```
## [1] 5
## [1] 10
## [1] 1
```

Supongamos que ahora definimos una nueva función *prueba2* como la sigueinte:

```
prueba2 <- function(x) {
    z <- 125980321654
    print(paste("Dentro de prueba2, z=",toString(z)))
    prueba1(z)
}
```

Veamos qué sucede al invocarla:

```
prueba2(5)
```

```
## [1] "Dentro de prueba2, z= 125980321654"
## [1] 125980321654
## [1] 251960643308
## [1] 1
```

El valor de z que recibe la función *prueba1* que aparece en el cuerpo de la función *prueba2* es $z = 125980321654$, número al que primero duplica, lo muestra, imprime también su doble, y finalmente recupera el valor de la variable libre z tal y como se definió en el entorno donde se creó la función *prueba1*.

Ejemplo:: Cubo de un número

```r
eleva_cubo <- function(n) {
    eleva_cuadrado <- function(n) {n*n}
    n*eleva_cuadrado(n)
    }
eleva_cubo(2)
```

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

Comentar finalmente que las funciones también pueden definirse con carácter recursivo:

Ejemplo: Factorial recursivo

```r
factorial_rec <- function(n){
    if (n==1){output <- 1}
    else {output <- n*factorial_rec(n-1)}
    output
}
factorial_rec(1)
```

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

```r
factorial_rec(2)
```

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

```r
factorial_rec(3)
```

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

```r
factorial_rec(4)
```

```
## [1] 24
```

```r
factorial_rec(5)
```

```
## [1] 120
```

```r
factorial_rec(10)
```

```
## [1] 3628800
```

Ejercicios

Ejercicio propuesto: Conversión de divisa

Construye una función que permita pasar de euros a dólares con eltipo de cambio actual.

Ejercicio propuesto: Serie de Leibniz

Construye una función que aproxime el valor de π tomando tantos términos de la serie de Leibniz

$$\sum_{n=0}^{\infty} \frac{(-1^n)}{2n+1} = \frac{\pi}{4}$$

como desee el usuario. Mide el tiempo que le lleva a R ejecutar dicha función para un millon de términos, por medio del comando *system.time()*.

Ejercicio propuesto: Números primos

Escribe un código que, para los 500 primeros núemros naturales, nos muestre por pantalla si son primos o compuestos

Ejercicio propeusto: Comisión del reaseguro

Supongamos que una compañía aseguradora tiene firmado un contrato de reaseguro proporcional, por el cual la cesionaria ha de devolver a la cedente al final del año una comisión que depende de la siniestralidad experimentada por esta última durante el ejercicio. Construye una función que permita obtener la comisión que habrá de recibir la aseguradora, si se pacta que ésta habrá de ser del 20% sobre las primas de reaseguro caundo la siniestralidad es inferior a 700.000€, del 5 % si excede los 2M€, y lineal en cualquier otro caso.

Ejercicio propuesto: Black-Scholes

Construye una función que permita obtener el precio de opciones europeas según la fórmula de Black-Scholes, tanto de compra como de venta:

$$c = SN(d_1) - X_e^{-rT} N(d_2)$$

:y:

$$p = X_e^{-rT} N(-d_2) - SN(-d_1)$$

siendo $d_1 = \frac{ln(S/X)+(r+\sigma^2/2)T}{\sigma\sqrt{T}}$ y $d_2 = d_1 - \sigma\sqrt{T}$. **Indicación**. La funcion de R *pnorm()* se corresponde con la función de distribución de una normal.|

Ejercicio propuesto: Potencia natural recursiona

Construye una función recursiova que permita calcular cualquier potencia natural de un número real.