


PROGRAMACIÓN CON R

Antonio Miñarro
aminarro@ub.edu



 Universitat de Barcelona
Departament d'Estadística

juny 2022

Esquema del tema

- 1 Introducción
- 2 Estructuras de Control de flujo
- 3 La familia apply
- 4 Funciones definidas por el usuario
- 5 Funciones genéricas y Clases
- 6 Depuración del código

Esquema del tema

- 1 Introducción
- 2 Estructuras de Control de flujo
- 3 La familia apply
- 4 Funciones definidas por el usuario
- 5 Funciones genéricas y Clases
- 6 Depuración del código

Introducción

- R es conocido como herramienta para analizar y graficar datos
- Pero también es un lenguaje de programación sencillo y muy versátil
- El hecho de que los programas no sean muy eficientes se ve compensado para la mayoría de usuarios por la facilidad de uso y la amigabilidad del entorno de programación
- La potencia de los programas se ve incrementada al poder acceder a las funciones incorporadas en R
- El usuario puede definir nuevas funciones que se adaptan a sus necesidades

Objetivos

- 1 Conocer las principales estructuras de control de código de R
- 2 Conocer la familia Apply
- 3 Aprender a crear funciones definidas por el usuario
- 4 Conocer herramientas de depuración del código

Necesidad de herramientas de programación

Imaginemos que queremos hacer un summary del conjunto de datos **mtcars** incluido en R.

```
> summary(mtcars)
```

mpg	cyl	disp	hp
Min. :10.40	Min. :4.000	Min. : 71.1	Min. : 52.0
1st Qu.:15.43	1st Qu.:4.000	1st Qu.:120.8	1st Qu.: 96.5
Median :19.20	Median :6.000	Median :196.3	Median :123.0
Mean :20.09	Mean :6.188	Mean :230.7	Mean :146.7
3rd Qu.:22.80	3rd Qu.:8.000	3rd Qu.:326.0	3rd Qu.:180.0
Max. :33.90	Max. :8.000	Max. :472.0	Max. :335.0

drat	wt	qsec	vs
Min. :2.760	Min. :1.513	Min. :14.50	Min. :0.0000
1st Qu.:3.080	1st Qu.:2.581	1st Qu.:16.89	1st Qu.:0.0000
Median :3.695	Median :3.325	Median :17.71	Median :0.0000
Mean :3.597	Mean :3.217	Mean :17.85	Mean :0.4375
3rd Qu.:3.920	3rd Qu.:3.610	3rd Qu.:18.90	3rd Qu.:1.0000
Max. :4.930	Max. :5.424	Max. :22.90	Max. :1.0000

am	gear	carb
Min. :0.0000	Min. :3.000	Min. :1.000
1st Qu.:0.0000	1st Qu.:3.000	1st Qu.:2.000
Median :0.0000	Median :4.000	Median :2.000
Mean :0.4062	Mean :3.688	Mean :2.812
3rd Qu.:1.0000	3rd Qu.:4.000	3rd Qu.:4.000
Max. :1.0000	Max. :5.000	Max. :8.000

Algunas variables son categóricas pero la descriptiva no lo recoge.

Necesidad de herramientas de programación (2)

Una solución: transformar en factores las variables categóricas.

```
> mtcars$cyl<-as.factor(mtcars$cyl)
> mtcars$gear<-as.factor(mtcars$gear)
> mtcars$carb<-as.factor(mtcars$carb)
> mtcars$am<-as.factor(mtcars$am)
> summary(mtcars)
```

mpg	cyl	disp	hp	drat	
Min. :10.40	4:11	Min. : 71.1	Min. : 52.0	Min. :2.760	
1st Qu.:15.43	6: 7	1st Qu.:120.8	1st Qu.: 96.5	1st Qu.:3.080	
Median :19.20	8:14	Median :196.3	Median :123.0	Median :3.695	
Mean :20.09		Mean :230.7	Mean :146.7	Mean :3.597	
3rd Qu.:22.80		3rd Qu.:326.0	3rd Qu.:180.0	3rd Qu.:3.920	
Max. :33.90		Max. :472.0	Max. :335.0	Max. :4.930	
wt	qsec	vs	am	gear	carb
Min. :1.513	Min. :14.50	Min. :0.0000	0:19	3:15	1: 7
1st Qu.:2.581	1st Qu.:16.89	1st Qu.:0.0000	1:13	4:12	2:10
Median :3.325	Median :17.71	Median :0.0000		5: 5	3: 3
Mean :3.217	Mean :17.85	Mean :0.4375			4:10
3rd Qu.:3.610	3rd Qu.:18.90	3rd Qu.:1.0000			6: 1
Max. :5.424	Max. :22.90	Max. :1.0000			8: 1

Ahora sí. Pero, y si hubiera muchas variables a las que aplicar la transformación! ¿Lo podemos hacer de una forma simplificada?

Bucle controlado por contador: Instrucción for

Sintaxis:

```
for (variable in secuencia) { instrucciones }
```

Ejemplo

```
> fibo<-c(1,1)
> for (i in 3:14) {
+   aux<-fibo[i-2]+fibo[i-1]
+   fibo<-c(fibo,aux)
+ }
> fibo

[1] 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

Aplicación a nuestro ejemplo

```
> data(mtcars) # Recargamos el dataframe original
> index<-c(2,9:11)
> for (i in index) {
+   mtcars[,i]<-as.factor(mtcars[,i])
+   print(paste('Transformada la variable',colnames(mtcars)[i]))
+ }
```

[1] "Transformada la variable cyl"
[1] "Transformada la variable am"
[1] "Transformada la variable gear"
[1] "Transformada la variable carb"

```
> summary(mtcars)
```

mpg	cyl	disp	hp	drat
Min. :10.40	4:11	Min. : 71.1	Min. : 52.0	Min. :2.760
1st Qu.:15.43	6: 7	1st Qu.:120.8	1st Qu.: 96.5	1st Qu.:3.080
Median :19.20	8:14	Median :196.3	Median :123.0	Median :3.695
Mean :20.09		Mean :230.7	Mean :146.7	Mean :3.597
3rd Qu.:22.80		3rd Qu.:326.0	3rd Qu.:180.0	3rd Qu.:3.920
Max. :33.90		Max. :472.0	Max. :335.0	Max. :4.930

wt	qsec	vs	am	gear	carb
Min. :1.513	Min. :14.50	Min. :0.0000	0:19	3:15	1: 7
1st Qu.:2.581	1st Qu.:16.89	1st Qu.:0.0000	1:13	4:12	2:10
Median :3.325	Median :17.71	Median :0.0000		5: 5	3: 3
Mean :3.217	Mean :17.85	Mean :0.4375			4:10
3rd Qu.:3.610	3rd Qu.:18.90	3rd Qu.:1.0000			6: 1
Max. :5.424	Max. :22.90	Max. :1.0000			8: 1

Bucle condicional: Instrucción while

Sintaxis:

```
while (condición) { instrucciones }
```

Ejemplo

```
> fibo<-c(1,1)
> while (length(fibo)<=12){
+   aux<-fibo[length(fibo)-1]+fibo[length(fibo)]
+   fibo<-c(fibo,aux)
+ }
> fibo

[1] 1 1 2 3 5 8 13 21 34 55 89 144 233
```

Operadores de comparación y lógicos

Operadores de comparación

- Igual: `==`
- No Igual: `!=`
- mayor/menor que: `><`
- mayor/menor o igual que: `>=<=`

Operadores lógicos

- Y: `&`
- O: `|`
- No: `!`
- Todos: `all(...)`
- Algún: `any(...)`
- O exclusivo: `xor(...)`

Ejecución condicional: Instrucción if

Opera sobre condiciones lógicas

Sintaxis:

```
if (cond=TRUE) { instrucciones 1 } [ else { instrucciones 2 } ]
```

Ejemplo

```
> x<-c(1,2,3,3,2,1)
> if (is.factor(x)) table(x) else mean(x)
[1] 2
> x<-factor(x)
> if (is.factor(x)) table(x) else mean(x)
x
1 2 3
2 2 2
```

Instrucciones comunes a todos los bucles

Instrucciones comunes a todos los bucles

- `break()`
para la ejecución del bucle.
- `next()`
para la iteración y avanza el contador.

Ejemplo

```
> for (i in 1:10){  
+   x<-runif(1,-1,1)  
+   if (x>0) y<-log(x) else next()  
+   cat(x, ' logaritme ',y,'\n')  
+ }
```

0.006029633 logaritme -5.111069
0.6481597 logaritme -0.4336182
0.08817717 logaritme -2.428407

Otra aplicación de la instrucción if

Recuperemos la estadística descriptiva del dataframe `mtcars`. La forma más eficiente de obtener las medias de las columnas es utilizar la función **`colMeans`**

```
> colMeans(mtcars)
```

Pero falla miserablemente! Probemos

```
> for (i in 1:ncol(mtcars)) {  
+   print(mean(mtcars[,i]))  
+ }
```

```
[1] 20.09062  
[1] NA  
[1] 230.7219  
[1] 146.6875  
[1] 3.596563  
[1] 3.21725  
[1] 17.84875  
[1] 0.4375  
[1] NA  
[1] NA  
[1] NA
```

En la siguiente diapositiva lo haremos mejor utilizando un `for` y un `if`.

Otra aplicación de la instrucción if (2)

```
> for (i in 1:ncol(mtcars)) {  
+ if (!is.factor(mtcars[,i])) print(mean(mtcars[,i]))  
+ }  
  
[1] 20.09062  
[1] 230.7219  
[1] 146.6875  
[1] 3.596563  
[1] 3.21725  
[1] 17.84875  
[1] 0.4375  
  
> for (i in 1:ncol(mtcars)) {  
+ if (!is.factor(mtcars[,i])) {  
+   cat('Variable:', colnames(mtcars)[i], '\t Promig:', mean(mtcars[,i]), '\n')  
+ } else cat('Variable:', colnames(mtcars)[i], '\t Es una variable factor.\n')  
+ }  
  
Variable: mpg           Promig: 20.09062  
Variable: cyl           Es una variable factor.  
Variable: disp          Promig: 230.7219  
Variable: hp            Promig: 146.6875  
Variable: drat           Promig: 3.596563  
Variable: wt            Promig: 3.21725  
Variable: qsec           Promig: 17.84875  
Variable: vs            Promig: 0.4375  
Variable: am            Es una variable factor.  
Variable: gear           Es una variable factor.  
Variable: carb          Es una variable factor.
```


Otra aplicación de la instrucción if (3)

```
> for (i in 1:ncol(mtcars)) {  
+   if (!is.factor(mtcars[,i])) {  
+     cat('Variable:', colnames(mtcars)[i], '\n Promig:', mean(mtcars[,i]), '\n\n')  
+   } else {  
+     cat('Variable:', colnames(mtcars)[i], '\n Taula:')  
+     print(table(mtcars[,i]))  
+     cat('\n\n')  
+   }  
+ }
```

```
Variable: mpg  
Promig: 20.09062
```

```
Variable: cyl  
Taula:  
 4  6  8  
11  7 14
```

```
Variable: disp  
Promig: 230.7219
```

```
Variable: hp  
Promig: 146.6875
```

```
Variable: drat  
Promig: 3.596563
```

```
Variable: wt  
Promig: 3.21725
```

Primera aproximación a las funciones personalizadas

R permite crear funciones personalizadas muy fácilmente

Estructura de una función

```
nombre.funcion<-function(argumentos){  
  secuencia de instrucciones  
}
```

Una vez cargada en memoria es suficiente invocar su nombre en el código

```
nombre.funcion(argumentos)
```

Aplicación a nuestro ejemplo

```
> resumen<-function(x){  
+   for (i in 1:ncol(x)) {  
+     if (!is.factor(x[,i])) {  
+       cat('Variable: ',colnames(x)[i], '\n Promig: ',mean(x[,i],na.rm=T), ', sd: ',sd(x[,i],na.rm=T), '\n\n')  
+     } else{  
+       cat('Variable: ',colnames(x)[i], '\n Taula: ')  
+       print(table(x[,i]))  
+       cat('\n\n')  
+     }  
+   }  
+ }  
+ }
```

```
> resumen(iris)  
  
Variable: Sepal.Length  
Promig: 5.843333 ,sd: 0.8280661  
  
Variable: Sepal.Width  
Promig: 3.057333 ,sd: 0.4358663  
  
Variable: Petal.Length  
Promig: 3.758 ,sd: 1.765298  
  
Variable: Petal.Width  
Promig: 1.199333 ,sd: 0.7622377  
  
Variable: Species  
Taula:  
  setosa versicolor  virginica  
    50         50         50
```

Ejercicio final de la sección

Crear un script que haga lo siguiente:

- Asigne una combinación de la primitiva (6 números entre 1 y 49) a una variable
- Vaya simulando de forma aleatoria sorteos de la primitiva y no pare hasta que tengamos al menos 3 aciertos en nuestra combinación
- Debe mostrar en cada paso la combinación ganadora y al final el número de sorteos necesarios
- ¿Os atreveis a hacerlo 10.000 veces para estimar el promedio de sorteos necesarios para conseguir 3 aciertos? (no querais escribir cada vez las combinaciones)

Ejercicio final de la sección

Crear un script que haga lo siguiente:

- Asigne una combinación de la primitiva (6 números entre 1 y 49) a una variable
- Vaya simulando de forma aleatoria sorteos de la primitiva y no pare hasta que tengamos al menos 3 aciertos en nuestra combinación
- Debe mostrar en cada paso la combinación ganadora y al final el número de sorteos necesarios
- ¿Os atreveis a hacerlo 10.000 veces para estimar el promedio de sorteos necesarios para conseguir 3 aciertos? (no querais escribir cada vez las combinaciones)

Ejercicio final de la sección

Crear un script que haga lo siguiente:

- Asigne una combinación de la primitiva (6 números entre 1 y 49) a una variable
- Vaya simulando de forma aleatoria sorteos de la primitiva y no pare hasta que tengamos al menos 3 aciertos en nuestra combinación
- Debe mostrar en cada paso la combinación ganadora y al final el número de sorteos necesarios
- ¿Os atreveis a hacerlo 10.000 veces para estimar el promedio de sorteos necesarios para conseguir 3 aciertos? (no querais escribir cada vez las combinaciones)

Ejercicio final de la sección

Crear un script que haga lo siguiente:

- Asigne una combinación de la primitiva (6 números entre 1 y 49) a una variable
- Vaya simulando de forma aleatoria sorteos de la primitiva y no pare hasta que tengamos al menos 3 aciertos en nuestra combinación
- Debe mostrar en cada paso la combinación ganadora y al final el número de sorteos necesarios
- ¿Os atreveis a hacerlo 10.000 veces para estimar el promedio de sorteos necesarios para conseguir 3 aciertos? (no querais escribir cada vez las combinaciones)

Esquema del tema

- 1 Introducción
- 2 Estructuras de Control de flujo
- 3 La familia apply**
- 4 Funciones definidas por el usuario
- 5 Funciones genéricas y Clases
- 6 Depuración del código

La familia "apply"

Cuál es la respuesta mas frecuente en los foros de internet a la pregunta

Q: ¿Como puedo utilizar un bucle para hacer (..cualquier acción..)?

A: De ninguna manera. Utiliza una de las funciones apply

Es verdad hasta cierto punto pero más que por el rendimiento porque permiten una programación más limpia y ordenada

La familia apply y funciones relacionadas

- 1 apply
- 2 sapply (lapply,vapply)
- 3 tapply
- 4 by
- 5 aggregate

La función apply

Devuelve un vector o lista obtenida al aplicar una función a las marginales (filas(1) o columnas(2)) de un array o matriz

Sintaxis:

```
apply(matriz,marginal,función)
```

Ejemplo

```
> m<-matrix(c(1:10,11:20),nrow=10,ncol=2)
> # promedio de las filas
> apply(m,1,mean)
[1] 6 7 8 9 10 11 12 13 14 15
> # promedio de las columnas
> apply(m,2,mean)
[1] 5.5 15.5
```

Nota: puede manejar sin problema arrays de dimensión superior a 2.

La función sapply

Devuelve un vector o matriz al aplicar una función a cada elemento de un objeto tipo data frame o lista. **(Para mí es una de las más útiles!)**

Sintaxis:

```
sapply(objeto, función)
```

Ejemplo

```
> l<-list(a=1:10,b=11:20)
> # suma de los valores de cada elemento
> sapply(l,sum)

  a  b 
55 155

> sapply(l,sum)[1]

a 
55
```

Ejemplo

```
> sapply(iris[,1:4],mean)

Sepal.Length Sepal.Width Petal.Length  Petal.Width 
  5.843333    3.057333    3.758000    1.199333 

> sapply(iris,is.numeric)

Sepal.Length Sepal.Width Petal.Length  Petal.Width   Species 
      TRUE      TRUE      TRUE      TRUE      FALSE
```

La función tapply

Aplica una función a cada conjunto de valores de un array determinados por los niveles de un determinado factor. Con ella empezamos el estudio de funciones que sirven para realizar descriptivas por niveles de un factor.

Sintaxis:

```
tapply(conjunto de valores,factor,función)
```

Ejemplo

```
> iris[1:2,]  
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species  
1           5.1          3.5          1.4          0.2  setosa  
2           4.9          3.0          1.4          0.2  setosa  
  
> tapply(iris$Sepal.Length,iris$Species,mean)  
  setosa versicolor  virginica  
  5.006    5.936    6.588
```

La función `by`

Aplica una función a cada subconjunto de un data frame definido por los niveles de uno o más factores (tapply aplicado a data frames).

Sintaxis:

```
by(data frame, factor, función)
```

Ejemplo

```
> by(iris[,1:4], iris$Species, colMeans)
```

```
iris$Species: setosa
```

Sepal.Length	Sepal.Width	Petal.Length	Petal.Width
5.006	3.428	1.462	0.246

```
-----  
iris$Species: versicolor
```

Sepal.Length	Sepal.Width	Petal.Length	Petal.Width
5.936	2.770	4.260	1.326

```
-----  
iris$Species: virginica
```

Sepal.Length	Sepal.Width	Petal.Length	Petal.Width
6.588	2.974	5.552	2.026

La función by (2)

Ejemplo

```
> by(iris, iris$Species, summary)
```

```
iris$Species: setosa
```

Sepal.Length	Sepal.Width	Petal.Length	Petal.Width
Min. :4.300	Min. :2.300	Min. :1.000	Min. :0.100
1st Qu.:4.800	1st Qu.:3.200	1st Qu.:1.400	1st Qu.:0.200
Median :5.000	Median :3.400	Median :1.500	Median :0.200
Mean :5.006	Mean :3.428	Mean :1.462	Mean :0.246
3rd Qu.:5.200	3rd Qu.:3.675	3rd Qu.:1.575	3rd Qu.:0.300
Max. :5.800	Max. :4.400	Max. :1.900	Max. :0.600

```
Species
```

```
setosa :50
versicolor: 0
virginica : 0
```

```
-----
iris$Species: versicolor
```

Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
Min. :4.900	Min. :2.000	Min. :3.00	Min. :1.000	setosa : 0
1st Qu.:5.600	1st Qu.:2.525	1st Qu.:4.00	1st Qu.:1.200	versicolor:50
Median :5.900	Median :2.800	Median :4.35	Median :1.300	virginica : 0
Mean :5.936	Mean :2.770	Mean :4.26	Mean :1.326	
3rd Qu.:6.300	3rd Qu.:3.000	3rd Qu.:4.60	3rd Qu.:1.500	
Max. :7.000	Max. :3.400	Max. :5.10	Max. :1.800	

```
-----
iris$Species: virginica
```

Sepal.Length	Sepal.Width	Petal.Length	Petal.Width
--------------	-------------	--------------	-------------

La función aggregate

Parecida a la anterior. Los factores tienen que ser una lista.

Sintaxis:

```
aggregate(data frame,factor(lista),función)
```

Ejemplo

```
> aggregate(iris[,1:4],list(iris$Species),mean)
```

	Group.1	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width
1	setosa	5.006	3.428	1.462	0.246
2	versicolor	5.936	2.770	4.260	1.326
3	virginica	6.588	2.974	5.552	2.026

```
> aggregate(iris[,1:4],list(iris$Species),summary)
```

	Group.1	Sepal.Length.Min.	Sepal.Length.1st Qu.	Sepal.Length.Median
1	setosa	4.300	4.800	5.000
2	versicolor	4.900	5.600	5.900
3	virginica	4.900	6.225	6.500

	Sepal.Length.Mean	Sepal.Length.3rd Qu.	Sepal.Length.Max.	Sepal.Width.Min.
1	5.006	5.200	5.800	2.300
2	5.936	6.300	7.000	2.000
3	6.588	6.900	7.900	2.200

	Sepal.Width.1st Qu.	Sepal.Width.Median	Sepal.Width.Mean	Sepal.Width.3rd Qu.
1	3.200	3.400	3.428	3.675
2	2.525	2.800	2.770	3.000
3	2.800	3.000	2.974	3.175

	Sepal.Width.Max.	Petal.Length.Min.	Petal.Length.1st Qu.	Petal.Length.Median
1	4.400	1.000	1.400	1.500
2	4.400	2.000	4.000	4.350

Resumen estadístico por niveles con by y aggregate

```
> by(mtcars[,1],mtcars$cyl,mean)
> aggregate(mtcars[,1],list(cilindres=mtcars$cyl),mean)
> aggregate(mtcars,list(cilindres=mtcars$cyl),mean)
> #by(mtcars,mtcars$cyl,mean)
> aggregate(mtcars,list(cilindres=mtcars$cyl,
+                   marxes=mtcars$gear),mean)
> aggregate(mpg~cyl+gear,data=mtcars,mean)
> aggregate(cbind(mpg,hp)~cyl+gear,data=mtcars,mean)
> aggregate(.~cyl+gear,data=mtcars,mean)
```


Combinación de la información de dos o más tablas

```
> t1<-aggregate(mpg~cyl+carb,FUN=mean,data=mtcars)
> t2<-aggregate(mpg~cyl+carb,FUN=length,data=mtcars)
> t1[,3]<-paste(t1[,3], ' (n=',t2[,3],') ',sep='')
> t1
```

	cyl	carb	mpg
1	4	1	27.58 (n=5)
2	6	1	19.75 (n=2)
3	4	2	25.9 (n=6)
4	8	2	17.15 (n=4)
5	8	3	16.3 (n=3)
6	6	4	19.75 (n=4)
7	8	4	13.15 (n=6)
8	6	6	19.7 (n=1)
9	8	8	15 (n=1)

Ejercicio final de la sección

- Generar una matriz de números aleatorios con 10.000 filas y 1.000 columnas
- Computar el tiempo que se tarda en sumar las 1.000 columnas
 - 1 Utilizando la función `apply`
 - 2 Utilizando un bucle `for` sobre las columnas
- Obtener una estadística descriptiva básica de las variables del conjunto de datos iris.
 - 1 Globalmente
 - 2 Separando por especies

Nota: generar los valores aleatorios con la función **`runif`** y utilizar la función **`system.time`** para calcular los tiempos de ejecución

Esquema del tema

- 1 Introducción
- 2 Estructuras de Control de flujo
- 3 La familia apply
- 4 Funciones definidas por el usuario**
- 5 Funciones genéricas y Clases
- 6 Depuración del código

Funciones en R

Una función es una colección de instrucciones agrupadas bajo un nombre, que realiza una tarea determinada y devuelve un resultado

Estructura general

```
> function.name <- function(argumentos){  
+   # Secuencia de instrucciones  
+   # que generan un resultado  
+   return(resultado)  
+ }
```

Ejemplo

```
> funPercent <- function (x){  
+   porcentaje<-round(x*100,digits=2)  
+   porcentaje<-paste(porcentaje,'%')  
+   return (porcentaje)  
+ }
```

Funciones en R

Una función es una colección de instrucciones agrupadas bajo un nombre, que realiza una tarea determinada y devuelve un resultado

Estructura general

```
> function.name <- function(argumentos){  
+   # Secuencia de instrucciones  
+   # que generan un resultado  
+   return(resultado)  
+ }
```

Ejemplo

```
> funPercent <- function (x){  
+   porcentaje<-round(x*100,digits=2)  
+   porcentaje<-paste(porcentaje,'%')  
+   return (porcentaje)  
+ }
```

Utilización de las funciones

Una vez cargada la función queda almacenada en memoria

```
> ls()
[1] "aux"          "cubo"          "fibo"          "funPercent"   "i"
[6] "index"        "l"             "l1"           "l2"           "m"
[11] "mtcars"       "n"            "resumen"      "t1"           "t2"
[16] "x"            "y"
```

y la podemos utilizar

```
> funPercent(0.236478)
[1] "23.65 %"

> prueba<-c(0.24,0.893443,1.254,0.6)
> funPercent(prueba)
[1] "24 %"      "89.34 %"  "125.4 %"  "60 %"
```

Jugando con los argumentos

- Los argumentos se identifican por el nombre o en su defecto por la posición que ocupan en la llamada a la función
- Podemos añadir tantos argumentos como queramos y podemos asignar valores por defecto a los argumentos

```
> funPercent <- function (x,dig=2){  
+   porcentaje<-round(x*100,digits=dig)  
+   porcentaje<-paste(porcentaje,'%')  
+   return (porcentaje)  
+ }
```

```
> funPercent(0.34767778377626)
```

```
[1] "34.77 %"
```

```
> funPercent(0.34767778377626,6)
```

```
[1] "34.767778 %"
```

```
> funPercent(dig=4,0.34767778377626)
```

```
[1] "34.7678 %"
```

Jugando con los argumentos

- Los argumentos se identifican por el nombre o en su defecto por la posición que ocupan en la llamada a la función
- Podemos añadir tantos argumentos como queramos y podemos asignar valores por defecto a los argumentos

```
> funPercent <- function (x,dig=2){  
+   porcentaje<-round(x*100,digits=dig)  
+   porcentaje<-paste(porcentaje, '%')  
+   return (porcentaje)  
+ }
```

```
> funPercent(0.34767778377626)  
[1] "34.77 %"  
  
> funPercent(0.34767778377626,6)  
[1] "34.767778 %"  
  
> funPercent(dig=4,0.34767778377626)  
[1] "34.7678 %"
```


Argumentos con puntos (...)

Es una forma de prever la posibilidad de pasar muchos tipos diferentes de argumentos a las funciones internas

```
> funPercent <- function (x,dig=2,...){  
+   porcentaje<-round(x*100,digits=dig)  
+   porcentaje<-paste(porcentaje,'% ',...)  
+   return (porcentaje)  
+ }
```

```
> funPercent(prueba)  
[1] "24 %"      "89.34 %" "125.4 %" "60 %"  
  
> funPercent(prueba,sep=' ')  
[1] "24%"      "89.34%" "125.4%" "60%"  
  
> funPercent(prueba,sep=' ',collapse='- ')  
[1] "24%-89.34%-125.4%-60%"
```

Argumentos con puntos (...)

Es una forma de prever la posibilidad de pasar muchos tipos diferentes de argumentos a las funciones internas

```
> funPercent <- function (x,dig=2,...){  
+   porcentaje<-round(x*100,digits=dig)  
+   porcentaje<-paste(porcentaje,'% ',...)  
+   return (porcentaje)  
+ }
```

```
> funPercent(prueba)  
[1] "24 %"      "89.34 %" "125.4 %" "60 %"  
  
> funPercent(prueba,sep=' ')  
[1] "24%"      "89.34%" "125.4%" "60%"  
  
> funPercent(prueba,sep=' ',collapse='- ')  
[1] "24%-89.34%-125.4%-60%"
```

La familia apply combinada con funciones propias

Vamos a plantearnos el siguiente problema. A partir de una matriz numérica queremos que todos los números se transformen en porcentajes con el símbolo % añadido. Existe la posibilidad de hacerlo de una tacada utilizando la función `apply` y una función definida por nosotros.

Ejemplo

```
> m<-matrix(runif(5*5),ncol=5)
> m
```

	[,1]	[,2]	[,3]	[,4]	[,5]
[1,]	0.370054899	0.1009011	0.06085891	0.7850633	0.4977677
[2,]	0.786574007	0.7225095	0.85923757	0.8822628	0.5245808
[3,]	0.003752992	0.3695690	0.84041612	0.2252996	0.3508259
[4,]	0.922897617	0.1461405	0.78125547	0.4137431	0.5134767
[5,]	0.444015591	0.6687831	0.93019036	0.4534919	0.1702404

```
> apply(m,c(1,2),function(x) paste(round(x,digits=3)*100,'%'))
```

	[,1]	[,2]	[,3]	[,4]	[,5]
[1,]	"37 %" "	"10.1 %" "	"6.1 %" "	"78.5 %" "	"49.8 %" "
[2,]	"78.7 %" "	"72.3 %" "	"85.9 %" "	"88.2 %" "	"52.5 %" "
[3,]	"0.4 %" "	"37 %" "	"84 %" "	"22.5 %" "	"35.1 %" "
[4,]	"92.3 %" "	"14.6 %" "	"78.1 %" "	"41.4 %" "	"51.3 %" "
[5,]	"44.4 %" "	"66.9 %" "	"93 %" "	"45.3 %" "	"17 %" "

Nota: verificar que ocurre si aplicamos la función sin `apply`

```
paste(round(m,digits=3)*100,'%')
```

```
)
```

La familia apply combinada con funciones propias

Vamos a plantearnos el siguiente problema. A partir de una matriz numérica queremos que todos los números se transformen en porcentajes con el símbolo % añadido. Existe la posibilidad de hacerlo de una tacada utilizando la función `apply` y una función definida por nosotros.

Ejemplo

```
> m<-matrix(runif(5*5),ncol=5)
> m
```

	[,1]	[,2]	[,3]	[,4]	[,5]
[1,]	0.370054899	0.1009011	0.06085891	0.7850633	0.4977677
[2,]	0.786574007	0.7225095	0.85923757	0.8822628	0.5245808
[3,]	0.003752992	0.3695690	0.84041612	0.2252996	0.3508259
[4,]	0.922897617	0.1461405	0.78125547	0.4137431	0.5134767
[5,]	0.444015591	0.6687831	0.93019036	0.4534919	0.1702404

```
> apply(m,c(1,2),function(x) paste(round(x,digits=3)*100,'%'))
```

	[,1]	[,2]	[,3]	[,4]	[,5]
[1,]	"37 %"	"10.1 %"	"6.1 %"	"78.5 %"	"49.8 %"
[2,]	"78.7 %"	"72.3 %"	"85.9 %"	"88.2 %"	"52.5 %"
[3,]	"0.4 %"	"37 %"	"84 %"	"22.5 %"	"35.1 %"
[4,]	"92.3 %"	"14.6 %"	"78.1 %"	"41.4 %"	"51.3 %"
[5,]	"44.4 %"	"66.9 %"	"93 %"	"45.3 %"	"17 %"

Nota: verificar que ocurre si aplicamos la función sin `apply`

```
paste(round(m,digits=3)*100,'%')
```

```
)
```

¡No tengais miedo de modificar funciones del sistema!

```
> prop.test(100,230,p=0.5)

1-sample proportions test with continuity correction

data: 100 out of 230, null probability 0.5
X-squared = 3.6565, df = 1, p-value = 0.05585
alternative hypothesis: true p is not equal to 0.5
95 percent confidence interval:
 0.3702057 0.5015743
sample estimates:
             p 
0.4347826
```

Nos puede servir para arreglar la salida con un estadístico Z en lugar de una X^2

```
1-sample proportions test with continuity correction

data: 100 out of 230, null probability 0.5
Z = 1.9122, df = 1, p-value = 0.05585
alternative hypothesis: true p is not equal to 0.5
95 percent confidence interval:
 0.3702057 0.5015743
sample estimates:
             p 
0.4347826
```

Esquema del tema

- 1 Introducción
- 2 Estructuras de Control de flujo
- 3 La familia apply
- 4 Funciones definidas por el usuario
- 5 Funciones genéricas y Clases**
- 6 Depuración del código

Funciones genéricas

¿Qué sucede si intentamos aplicar nuestra función de porcentajes a una cadena de texto?

```
> funPercent('a')
```

```
Error in x * 100 : non-numeric argument to binary operator
```

Podemos crear una función modificada que acepte cadenas de texto

```
> funPercent.character <- function (x){  
+   porcentaje<-paste(x,'%')  
+   return (porcentaje)  
+ }  
  
> funPercent.character('a')  
[1] "a %"
```

Funciones genéricas

¿Qué sucede si intentamos aplicar nuestra función de porcentajes a una cadena de texto?

```
> funPercent('a')
```

```
Error in x * 100 : non-numeric argument to binary operator
```

Podemos crear una función modificada que acepte cadenas de texto

```
> funPercent.character <- function (x){  
+   porcentaje<-paste(x,'%')  
+   return (porcentaje)  
+ }  
  
> funPercent.character('a')  
[1] "a %"
```


Creación de la función genérica

En primer lugar copiamos la función original en una nueva función

```
> funPercent.numeric<-funPercent
```

Y ahora generamos la función genérica y la podemos utilizar

```
> funPercent <- function(x,...){  
+   UseMethod('funPercent')  
+ }  
> funPercent(0.34)  
[1] "34 %"  
  
> funPercent('Hola')  
[1] "Hola %"
```

Creación de la función genérica

En primer lugar copiamos la función original en una nueva función

```
> funPercent.numeric<-funPercent
```

Y ahora generamos la función genérica y la podemos utilizar

```
> funPercent <- function(x,...){  
+   UseMethod('funPercent')  
+ }  
> funPercent(0.34)  
[1] "34 %"  
> funPercent('Hola')  
[1] "Hola %"
```

Consulta de los métodos

Podemos saber que métodos están asociados a una función genérica

Consulta del método

```
> methods(funPercent)

[1] funPercent.character funPercent.numeric
see '?methods' for accessing help and source code
```

Vamos a probar con la función **print**

```
> print.methods<-methods(print)
> length(print.methods)

[1] 188

> head(print.methods,10)

[1] "print.acf"           "print.anova"
[3] "print.aov"           "print.aovlist"
[5] "print.ar"            "print.Arima"
[7] "print.arima0"        "print.AsIs"
[9] "print.aspell"        "print.aspell_inspect_context"
```

Creación de clases (S3 Old-Style)

Imaginemos que queremos crear una clase de objetos denominada curso que sirva para contener datos de los alumnos

Definición de la clase

```
> al<-list(nombre='Pilar',facultad='Biologia',edad=25,mujer=T)
> class(al)<-'alumno'
> al

$nombre
[1] "Pilar"

$facultad
[1] "Biologia"

$edad
[1] 25

$mujer
[1] TRUE

attr(,"class")
[1] "alumno"
> al[[1]]
[1] "Pilar"
```

Creación de un método específico para la clase

Queremos crear un método de la función **print** específico para la clase **alumno**

```
> print.alumno<-function(obj){  
+   cat(obj$nombre, '\n')  
+   cat('Facultad:', obj$facultad, '\n')  
+   cat('Edad:', obj$edad, '\n')  
+   sexo<-ifelse(obj$mujer, 'Mujer', 'Varón')  
+   cat('Sexo:', sexo, '\n')  
+ }
```

```
> al  
Pilar  
Facultad: Biologia  
Edad: 25  
Sexo: Mujer
```

Creación de un método específico para la clase

Queremos crear un método de la función **print** específico para la clase **alumno**

```
> print.alumno<-function(obj){  
+   cat(obj$nombre, '\n')  
+   cat('Facultad:', obj$facultad, '\n')  
+   cat('Edad:', obj$edad, '\n')  
+   sexo<-ifelse(obj$mujer, 'Mujer', 'Varón')  
+   cat('Sexo:', sexo, '\n')  
+ }
```

```
> al  
Pilar  
Facultad: Biologia  
Edad: 25  
Sexo: Mujer
```

Creación de un método específico (2)

Evidentemente ahora una consulta a los métodos de **print** incluye el nuevo método **print.alumno**

```
> head(methods(print))  
[1] "print.acf"      "print.alumno"   "print.anova"    "print.aov"  
[5] "print.aovlist"  "print.ar"
```

Creación de clases (S4 New-Style)

Queda fuera de los objetivos del presente curso pero es importante saber que existe un nuevo sistema de objetos denominado **S4** basado en la programación orientada a objetos (OOP). Tan solo mostramos como ejemplo cómo se define una clase S4 y su estructura. Para definir una clase ahora utilizamos la instrucción **setClass**

```
> setClass('alumno',  
+   representation(  
+     nombre='character',  
+     facultad='character',  
+     edad='numeric',  
+     mujer='logical')  
+ )
```

Para generar un nuevo elemento de la clase

```
> pili<-new('alumno',nombre='Pilar',facultad='Biologia',edad=23,mujer=T)  
> pili  
  
An object of class "alumno"  
Slot "nombre":  
[1] "Pilar"  
  
Slot "facultad":  
[1] "Biologia"  
  
Slot "edad":  
[1] 23  
  
Slot "mujer":  
[1] TRUE
```


Esquema del tema

- 1 Introducción
- 2 Estructuras de Control de flujo
- 3 La familia apply
- 4 Funciones definidas por el usuario
- 5 Funciones genéricas y Clases
- 6 Depuración del código

Tipos de error

- 1 Errores de sintaxis o semánticos
- 2 Errores lógicos

Tipos de error

- 1 Errores de sintaxis o semánticos
- 2 Errores lógicos

Mensajes de Error y Advertencias (Warnings)

- Warnings

El código se ejecuta hasta el final y cuando acaba se muestra el aviso

Ejemplo

```
x<- -3:3
y<-sqrt(x)
Warning message: In sqrt(x) : NaNs produced
y
[1]      NaN      NaN      NaN 0.000000 1.000000 1.414214 1.732051
```

- Error

El código se para inmediatamente y se muestra el aviso de error

Ejemplo

```
x<-letters[1:10]
rm(y)
y<-sqrt(x)
Error in sqrt(x) : non-numeric argument to mathematical function
y
Error: object 'y' not found
```

Generación de mensajes personalizados

Ejemplo

```
> for (i in 1:10){  
+   x<-runif(1,-1,1)  
+   if (x>0) y<-log(x) else{  
+     warning(paste('Ha aparecido un número negativo:',x,' en la iteración',i))  
+     next()  
+   }  
+   cat(x, ' logaritme ',y,'\n')  
+ }
```

0.8311314	logaritme	-0.1849674
0.3474099	logaritme	-1.05725
0.4251701	logaritme	-0.8552658
0.8530392	logaritme	-0.1589497
0.4951704	logaritme	-0.7028534
0.6310448	logaritme	-0.4603785
0.5694561	logaritme	-0.5630735
0.2639434	logaritme	-1.332021
0.765857	logaritme	-0.2667598

Generación de mensajes personalizados (2)

Para generar un mensaje de error y parar el código simplemente sería necesario substituir la función **warning** por la función **stop**.

Ejemplo

```
> for (i in 1:10){  
+   x<-runif(1,-1,1)  
+   if (x>0) y<-log(x) else{  
+     stop(paste('Ha aparecido un número negativo:',x,' en la iteración',i))  
+     # break() no necesario  
+   }  
+   cat(x,' logaritme ',y,'\n')
```

"Bug Hunting"

Los errores y advertencias nos informan de qué línea genera el aviso, pero no es garantía de que los problemas se hayan generado en esa línea.

Funciones para el ejemplo

```
> # Función principal
> geom<-function(x){
+   mos<-mostra(x)
+   geomean<-geomean(mos)
+   res<-paste('La media geométrica es:',geomean)
+   return(res)
+ }
> # Función que genera una muestra de tamaño x
> mostra<-function(x){
+   mos<-rnorm(x)
+   return(mos)
+ }
> # Función que calcula la media geométrica
> geomean<-function(x){
+   meangeo<-exp(mean(log(x)))
+   if(meangeo>1) result<-'GM > 1' else result<-'GM < 1'
+   return(result)
+ }
```

La función traceback

Si llamamos a la función geom anterior obtenemos

```
geom(5)
Error in if (geomean > 1) result <- "GM > 1" else result <- "GM < 1" :
  missing value where TRUE/FALSE needed
In addition: Warning message:
In log(x) : NaNs produced
```

Para saber realmente dónde se ha producido el error

```
> traceback()
2: geomean(mos) at #3
1: geom(5)
```


La función debug

```
> debug(geomean)

> geom(5)
debugging in: geomean(mos)
debug at #1:
meangeo <- exp(mean(log(x)))
if (meangeo > 1)
  result <- "GM > 1"
else result <- "GM < 1"
return(result)

Browse[2]> n
debug at #2: meangeo <- exp(mean(log(x)))
Browse[2]> n
debug at #3: if (meangeo > 1) result <- "GM > 1" else result <- "GM < 1"
```

"tags" para la función debug

n -> línea adelante c-> completar el código Q-> salir del browser

Para que no se vuelva a ejecutar el browser

```
undebug(geomean)
```

Midiendo el rendimiento

Ya hemos mencionado algún momento la función **system.time**

```
> m<-matrix(runif(200000),20000)
> system.time(apply(m,1,sum))
   user  system elapsed 
 0.03    0.00    0.03
```

Existen formas más eficientes de comparar el rendimiento de, por ejemplo, dos funciones o dos algoritmos, evitando artefactos debidos a la maquinaria. Uno de los packages que nos puede ayudar es **rbenchmark**

```
> install.packages('rbenchmark')
> library(rbenchmark)
```

Utilización de **benchmark**

Vamos a comparar dos maneras de calcular las sumas de las filas de una matriz

```
> f0<-function(x) apply(x,1,sum)
> f1<-function(x) rowSums(x)
> benchmark(f0(m),f1(m),columns=c('test','elapsed','relative'),replications=50)

  test elapsed relative
1 f0(m)   1.39    69.5
2 f1(m)   0.02     1.0
```

Utilización de **benchmark**

Vamos a comparar dos maneras de calcular las sumas de las filas de una matriz

```
> f0<-function(x) apply(x,1,sum)
> f1<-function(x) rowSums(x)
> benchmark(f0(m),f1(m),columns=c('test','elapsed','relative'),replications=50)
```

	test	elapsed	relative
1	f0(m)	1.39	69.5
2	f1(m)	0.02	1.0

Comparación de resultados

Evidentemente saber que un algoritmo es más rápido no sirve de nada si no produce los mismos resultados. Para comparar resultados tenemos diversas posibilidades

```
> identical(f0(m),f1(m))  
[1] TRUE  
> identical(c(1,1),c(1,0.9999))  
[1] FALSE  
> all.equal(c(1,1),c(1,0.9999))  
[1] "Mean relative difference: 1e-04"  
> all.equal(c(1,1),c(1,0.9999),tolerance=0.0001)  
[1] TRUE  
> all.equal(c(x=1,y=1),c(z=1,g=1))  
[1] "Names: 2 string mismatches"  
> all.equal(c(x=1,y=1),c(z=1,g=1),check.attributes=F)  
[1] TRUE
```

Ejercicio final de la sesión

- Escribir una función `MtTest` que reciba como argumentos:
 - Un data frame
 - El nombre en carácter de una de las columnas del data frame que debe ser un factor con dos niveles
 - Opcionalmente un valor de nivel de significación (por defecto 0.05)
 - Dejar abierta la posibilidad de otros argumentos que pueda utilizar la función `t.test`
- La función debe comprobar y pararse emitiendo un mensaje de error si:
 - La variable suministrada no corresponde a un factor.
 - Siendo factor el número de niveles no es igual a 2.
- La función debe devolver para cada columna numérica del data frame:
 - El p-valor de la prueba t de Student para comparar los dos niveles de la variable factor.
 - El resultado del contraste en forma de: 'H0' o 'H1'.



FIN DE LA SESIÓN