Introducción al software estadístico

Módulo II

Nicolás Schmidt

nschmidt@cienciassociales.edu.uy

Departamento de Ciencia Política
Unidad de Métodos y Acceso a Datos
Facultad de Ciencias Sociales
Universidad de la República

Estructura de la presentación

1. Programación
   - Orientada a Objetos
   - Funcional

2. Identificadores y coercionadores

Estructura de la presentación

1. Programación
   - Orientada a Objetos
   - Funcional
2. Identificadores y coercionadores

ADVERTENCIA

Esta es una introducción breve, simplificada e incompleta sobre programación orientada a objetos y programación funcional. No vamos a abordar las clases de objetos S3 y S4. Solo vamos a abordar los tópicos que sirvan para comprender los tipos y las clases de objetos más comunes en R para un uso básico.

"To understand computations in R, two slogans are helpful:

Everything that exists is an object.
Everything that happens is a function call."

John Chambers

OOP

La programación orientada a objetos (simplificando mucho) refiere a que dada una estructura de datos el programador define los tipos de datos y las operaciones que se pueden realizar con ellos. Los conceptos de *clase* y *método* son clave en esta forma de programar.

clase: Un objeto en el lenguaje debe ser una instancia de alguna clase.

método: Los cálculos se llevan a cabo a través de métodos. Los métodos son funciones para llevar a cabo cálculos específicos en objetos, generalmente de una clase específica.

De esta manera los objetos son de un tipo (`typeof()`), pertenecen a una clase (`class()`) con determinados atributos (`attributes()`) y tienen una estructura (`str()`) específica.

Objetos

⇒ En R se crea un objeto con el asignador ‘‘<-’’. También se puede usar ‘‘=’’ pero no es recomendable.

```
nombre.del.objeto <- contenido.del.objeto
```

Palabras reservadas				
if	break	next	TRUE	NaN
else	for	NULL	Inf	NA
FALSE	function	repeat	while	in

⇒ **Importante**: el nombre de un objeto no puede iniciar con números, ni signos, ni puede tener espacios en blanco.

⇒ **Relevante**: R es sensible a minúsculas y mayúsculas.

⇒ **Sugerencia**: evitar nombrar objetos con nombres de funciones existentes.

Ejemplo:

```
# OPCION 1
objeto.1 <- matrix(1:4, 2, 2)  # Guardo una matriz en el objeto llamado 'objeto.1'
objeto.1                       # Imprimo el objeto escribiendo su nombre

##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4

# OPCION 2
matrix(1:4, 2, 2) -> objeto.2  # El asignador esta en el otro sentido.
print(objeto.2)                # Imprimo el objeto con la función print()

##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4

# OPCION 3
(objeto.3 = matrix(1:4, 2, 2)) # Uso el signo '=' para asignar datos a un objeto

##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4

                               # Imprimo el objeto usando paréntesis
```

Estructura, clase y atributos de un objeto

Ejemplo:

```
class(objeto.1)

## [1] "matrix"

typeof(objeto.1)

## [1] "integer"

attributes(objeto.1)

## $dim
## [1] 2 2

str(objeto.1)

##  int [1:2, 1:2] 1 2 3 4
```

Posible fuente de errores y confusiones

Ejemplo: no dejar espacio en el asignador '<-'

```
x <- 1:10
x[2] < - 5  #no dejar espacio entre '<' y "-" sino se esta haciendo una comparación


## [1] FALSE
```

Ejemplo: uso del asignador '<-'

```
if(a <- 1L == 1L) print("Hola mundo")

## [1] "Hola mundo"
```

Ejemplo: uso del asignador '=' en la misma sentencia

```
if(a = 1 == 1) print("Hola mundo")

## Error: <text>:1:6: unexpected '='
## 1: if(a =
##          ^
```

Utilidad de trabajar con objetos

Para un uso inicial se destacan:

- Realizar operaciones sin imprimir en consola el resultado.
- Extraer una parte del objeto y usarla como resultado o para hacer otras operaciones (modificabilidad).
- Ordena mejor la secuencia de código.
- Múltiples copias de un objeto.

Tipos de objetos: `typeof()`

Los tipos de objetos más frecuentes son estos:

typeof*	Descripción	Ejemplo
`NULL`	NULO	`a <- NULL`
`closure`	Una función	`a <- function(x){x*pi}`
`logical`	un vector que contiene valores lógicos	`a <- c(TRUE, FALSE, TRUE)`
`integer`	un vector que contiene valores enteros	`a <- c(1:10)`
`double`	un vector que contiene valores reales	`a <- seq(1:10, 0.3)`
`complex`	un vector que contiene valores complejos	`a <- sqrt(-17+0i)`
`character`	un vector que contiene valores de caracteres	`a <- letters[1:5]`
`list`	una lista	`a <- list(b=1:10, c="j")`

*Nota: en este link está la lista completa de posibles respuestas de la función `typeof()`.

Estructuras de datos en R

Dimensiones	Homogéneo	Heterogéneo
1	Vector atómico	List
2	Matrix	Data Frame
n	Array	

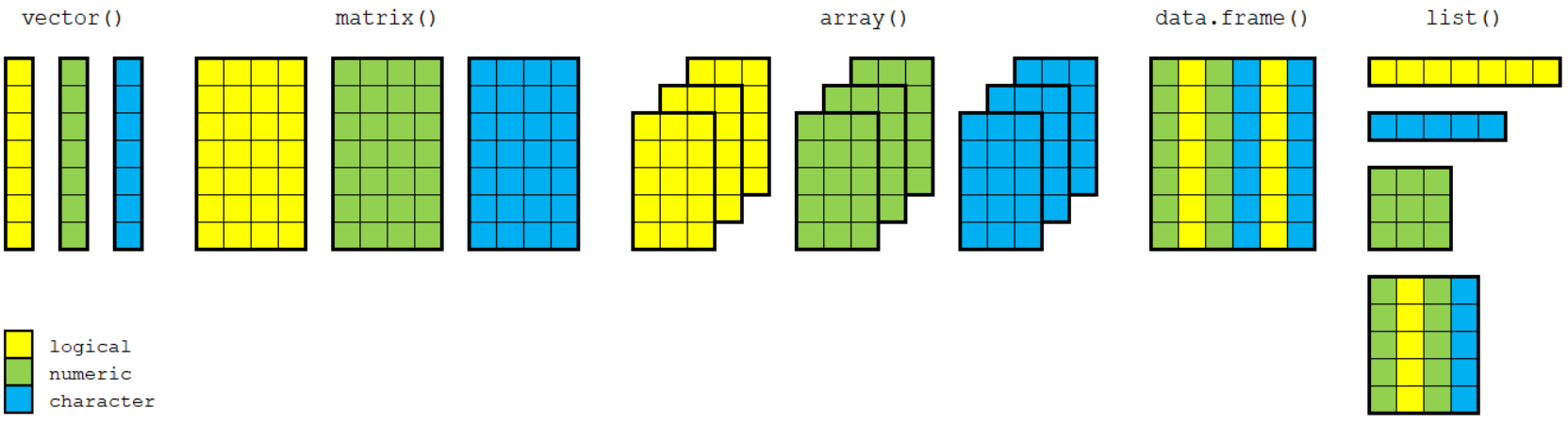
Fuente: Wickham, Hadley. *Advanced R*

Estructuras de datos en R

La estructura fundamental en R es el `vector`. El resto de las estructuras se pueden pensar como combinaciones de vectores.

- → `matrix()` y `array()`: una matriz o un arreglo es un vector con el atributo `dim()`(dimensión).
- → `data.frame()`: uno o más vectores del mismo largo (`length()`) que en el caso de un marco de datos sería con la misma cantidad de filas (`nrow()`)
- → `list()`: un lista es un vector genérico. Cada elemento de una lista puede ser de distinto tipo.

Estructuras de datos en R



Fuente: adaptación de Friendly, Michael, Meyer, David (2015)

Programación Funcional

R es también (y principalmente) un lenguaje de programación funcional. Esto significa que tiene un conjunto de herramientas para la creación, modificación y uso de funciones que potencian la utilidad del lenguaje.

Una función puede ser asignada a un objeto, usada por otra función o puede ser el resultado de otra función.

En esta parte del curso lo que interesa destacar es que la mayoría de las operaciones que se van a realizar son llamando a una función.

> Lo importante es identificar qué *tipo* y *clase* de objeto de entrada requiere la función y qué *tipo* y *clase* de objeto devuelve la función.

Ejemplo 1:

```
datos <- data.frame(x = 1:5, y = 5:1, z = letters[1:5])
print(datos)
```

```
##   x y z
## 1 1 5 a
## 2 2 4 b
## 3 3 3 c
## 4 4 2 d
## 5 5 1 e
```

```
mean(datos)
```

```
## Warning in mean.default(datos): argument is not numeric or logical: returning
NA
```

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

```
mean(datos[,1])
```

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

Ejemplo 2:

```
apply(datos, 2, sum)

## Error in FUN(newX[, i], ...): invalid 'type' (character) of argument

apply(datos[, 1:2], 2, sum)

##  x  y
## 15 15

sumaCol <- apply(datos[, 1:2], 2, sum)
print(sumaCol)

##  x  y
## 15 15
```

Versatilidad de la Programación Funcional

Ejemplo 3:

```
y <- 1:10
mean(y)          # Uso una función

## [1] 5.5

m <- mean(y)     # Guardo la salida de una función
m

## [1] 5.5

media <- mean    # asigno una función (guardo todo el ambiente de la función)
media(y)

## [1] 5.5
```

Versatilidad de la Programación Funcional

Ejemplo 4:

```
lista <- list(y, y*10, y^2)

sapply(lista, mean)

## [1]  5.5 55.0 38.5

sapply(lista, sum)

## [1]  55 550 385

sapply(lista, function(x){median(x) + 1})

## [1]  6.5 56.0 31.5
```

Versatilidad de la Programación Funcional

Agreguemos complejidad: una función que aplica distintas funciones a distintos objetos.

Ejemplo 5:

```
print(y)

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

funciones <- c("mean", "sum", "median")

purrr::invoke_map(funciones, list(list(y), list(y*10),list( y^2)))

## [[1]]
## [1] 5.5
##
## [[2]]
## [1] 550
##
## [[3]]
## [1] 30.5
```

Estructura de la presentación

`is.*` y `as.*`

En R hay un conjunto de funciones que permiten que un objeto de determinado tipo cambie a otro tipo diferente (*coerción*). Este tipo de operaciones adquieren utilidad dado que las funciones que vienen en R y las que se puedan usar de algún paquete requieren un *tipo* y una *clase* especifica de entrada.

Como hemos visto, las estructuras de objeto más usadas en R son en algún punto combinaciones de vectores. Esto hace que algunas conversiones sean simples.

Los *identificadores* (`is.*`) a diferencia de los *coercionadores* (`as.*`) tienen gran utilidad para programar funciones.

'is.*' y 'as.*' más utilizados

Tipo	Verificación	Cambio
vector	is.vector()	as.vector()
list	is.list()	as.list()
data.frame	is.data.frame()	as.data.frame()
matrix	is.matrix()	as.matrix()
logical	is.logical()	as.logical()
factor	is.factor()	as.factor()
character	is.character()	as.character()
numeric	is.numeric()	as.numeric()
double	is.double()	as.double()

Para ver todas las opciones disponibles

```
ls(patter = "^is.", baseenv())
ls(patter = "^as.", baseenv())
```

Ejemplo 1:

```
datos <- data.frame(x = 1, y = 1:3, z = c(2, 0, 1))
datos
```

```
##   x y z
## 1 1 1 2
## 2 1 2 0
## 3 1 3 1
```

```
det(datos)
```

```
## Error in UseMethod("determinant"): no applicable method for 'determinant'
applied to an object of class "data.frame"
```

```
datos.1 <- as.matrix(datos)
datos.1
```

```
##      x y z
## [1,] 1 1 2
## [2,] 1 2 0
## [3,] 1 3 1
```

```
det(datos.1)
```

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

Ejemplo 2:

```
datos                          # miro el data.frame creado

##   x y z
## 1 1 1 2
## 2 1 2 0
## 3 1 3 1

class(datos)                   # chequeo la clase del objeto

## [1] "data.frame"

t(datos)                       # hago una operación matricial con un 'data.frame'

##   [,1] [,2] [,3]
## x    1    1    1
## y    1    2    3
## z    2    0    1

transpuesta <- t(datos)        # guardo la operación en el objeto 'transpuesta'
class(transpuesta)             # chequeo la clase del objeto

## [1] "matrix"
```