

Bloque 5: Funciones

Las funciones son el pilar de la programación modular. En R se definen con `function()` y son objetos de primera clase (pueden pasarse como argumentos a otras funciones).

5.1 Anatomía de una función

```
# Sintaxis basica
nombre_funcion <- function(arg1, arg2 = valor_por_defecto) {
  # Cuerpo de la funcion
  resultado <- arg1 + arg2
  return(resultado) # valor de retorno
}

# Llamadas
nombre_funcion(3, 7)      # usando posicion
nombre_funcion(arg1 = 3)  # arg2 usa el valor por defecto
```

Ejercicio 13: Validación de usuario

Una función que valida si un nombre de usuario es correcto: menos de 10 caracteres, sin la secuencia 'ri', y que no sea ni 'Enrique' ni 'Ana'.

```
usuario_valido <- function(nombre) {
  val_long      <- nchar(nombre) < 10
  val_ri        <- !grepl("ri", nombre, ignore.case = TRUE)
  val_ocupados  <- !(nombre %in% c("Enrique", "Ana"))
  return(val_long & val_ri & val_ocupados)
}

usuario_valido("Enrique")  # FALSE (nombre prohibido)
usuario_valido("Carlos")   # TRUE
usuario_valido("Francisca") # FALSE (>= 10 caracteres)
usuario_valido("Mario")    # FALSE (contiene 'ri')
```

Explicación: `grepl()` con `ignore.case=TRUE` busca sin distinguir mayúsculas. `%in%` comprueba si el valor está en el vector. `&` combina condiciones lógicas.

Ejercicio 14: Calculadora con switch

Función calculadora que recibe dos números y una operación ('suma','resta','multiplica','divide').

```
calculadora <- function(a, b, op) {  
  switch(op,  
    suma      = a + b,  
    resta     = a - b,  
    multiplica = a * b,  
    divide    = {  
      if (b == 0) stop("No se puede dividir por cero")  
      a / b  
    },  
    stop(paste("Operacion desconocida:", op))  
  )  
}  
  
calculadora(10, 3, "suma")      # 13  
calculadora(10, 3, "divide")   # 3.333...  
calculadora(10, 0, "divide")   # Error
```

Explicación: switch() es más limpio que encadenar if/else para seleccionar entre varias opciones. El ultimo argumento sin nombre actúa como 'default'.

Ejercicio 15: Función con múltiples retornos (lista)

Crea una función que reciba un vector numérico y devuelva una lista con sus estadísticas principales.

```
estadisticas <- function(v) {  
  list(  
    n      = length(v),  
    media  = mean(v),  
    mediana = median(v),  
    sd     = sd(v),  
    minimo = min(v),  
    maximo = max(v),  
    rango  = max(v) - min(v)  
  )  
}  
  
set.seed(1)  
datos <- rnorm(100, mean = 50, sd = 10)  
res <- estadisticas(datos)  
  
res$media    # acceso por nombre  
res[["sd"]]  # acceso alternativo
```

Explicación: Las listas en R pueden contener elementos de distintos tipos. Son perfectas para devolver múltiples resultados desde una función.

Ejercicio 16: Vectorizar una función con Vectorize()

La función `es_par()` funciona con un solo número. Vectorízala para que funcione con vectores enteros.

```
# Version escalar (solo funciona con un numero)
es_par <- function(n) {
  if (n %% 2 == 0) "par" else "impar"
}

es_par(4)    # "par"
# es_par(1:5) # da advertencia o resultado incorrecto

# Vectorizamos
es_par_v <- Vectorize(es_par)
es_par_v(1:10)
# "impar" "par" "impar" "par" ...

# Alternativa nativa (mas eficiente)
ifelse(1:10 %% 2 == 0, "par", "impar")
```

Explicación: `Vectorize()` envuelve la función en un `sapply` internamente. Para casos simples, `ifelse()` vectorizado es más rápido, pero `Vectorize()` es útil para funciones complejas con `if/else`.

Bloque 6: Algoritmos — Números Primos

Los números primos son un clásico para practicar algoritmos. Veremos varias aproximaciones, de la más fácil a la más optimizada.

Ejercicio 17: Comprobar si un número es primo

Escribe una función que determine si un número entero positivo es primo.

```
es_primo <- function(n) {  
  if (n < 2) return(FALSE)  
  if (n == 2) return(TRUE)  
  if (n %% 2 == 0) return(FALSE)  
  # Solo comprobamos hasta la raíz cuadrada  
  divisores <- seq(3, sqrt(n), by = 2)  
  if (length(divisores) == 0) return(TRUE)  
  return(all(n %% divisores != 0))  
}  
  
es_primo(2)    # TRUE  
es_primo(7)    # TRUE  
es_primo(12)   # FALSE  
es_primo(97)   # TRUE
```

Explicación: Solo necesitamos comprobar divisores hasta $\sqrt{n}$: si n tiene un factor mayor que su raíz, el otro factor sería menor que ella. Descartar pares desde el inicio duplica la velocidad.

Ejercicio 18: Medida de Stanton

Dado un vector aleatorio de valores entre 1 y 7, calcula la Medida de Stanton: cuenta cuantos 1 hay, ese número es n ; la medida es cuantos n hay en el vector.

```
set.seed(1341)  
vs <- sample(1:7, size = 20, replace = TRUE)  
vs  
  
# Paso 1: cuantos 1 hay  
n <- sum(vs == 1)  
cat("Numero de 1s:", n, "\n")  
  
# Paso 2: cuantos 'n' hay  
stanton <- sum(vs == n)  
cat("Medida de Stanton:", stanton, "\n")
```

Explicación: `sum()` sobre un vector lógico cuenta los TRUE. `vs==1` genera un vector TRUE/FALSE; `sum()` cuenta los TRUE. La Medida de Stanton es una curiosidad matemática de autorreferencia.

Ejercicio 19: Todos los primos hasta N

Encuentra todos los números primos menores o iguales a 1000 usando la Criba de Eratóstenes.

```
criba_eratostenes <- function(N) {  
  if (N < 2) return(integer(0))  
  es_primo <- rep(TRUE, N)  
  es_primo[1] <- FALSE  
  for (i in 2:floor(sqrt(N))) {  
    if (es_primo[i]) {  
      # Marcar todos los multiples como no primos  
      multiplos <- seq(i*i, N, by = i)  
      es_primo[multiplos] <- FALSE  
    }  
  }  
  which(es_primo)  
}  
  
primos <- criba_eratostenes(1000)  
length(primos) # 168 primos hasta 1000  
head(primos, 10) # 2 3 5 7 11 13 17 19 23 29
```

Explicación: La Criba de Eratóstenes es mucho más rápida que comprobar cada número individualmente. En lugar de dividir, 'tacha' los múltiplos de cada primo. `which()` devuelve los índices donde el vector es TRUE.