Unidad III pre

Primeros pasos en programación en R

El dominio de la programación en R es una competencia esencial en la formación de un estudiante de Magíster en Ciencia de Datos. En este capítulo, aprenderemos a controlar el flujo de ejecución mediante estructuras como `for()`, `while()`, `if()` y `repeat()`, aplicables en tareas como la iteración sobre datos, la evaluación condicional de modelos y la automatización de procesos analíticos. Se introducirá la definición de funciones propias con `function()`, el uso de `replicate()` para simulaciones estadísticas, y técnicas de depuración con `debug()` y `browser()` para asegurar la corrección del código. Al finalizar este capítulo, el estudiante será capaz de construir scripts modulares, eficientes y mantenibles, fundamentales para resolver problemas complejos de clasificación, predicción o procesamiento masivo de datos.

SESION 1: Control del flujo de operaciones por medio de loops

Uso de `for`

para (`for`) cada elemento_`i` en una lista:

    hacer algo con elemento_`i`

Ejemplo de n!

El factorial n! cuenta cuántas formas diferentes se pueden ordenar n objetos diferentes. Se define como:

$$n! = 1 \cdot 2 \cdot 3 \cdots (n-1) \cdot n$$

Una forma de calcularlo sería usar una declaración `for()`. Por ejemplo, podríamos encontrar el valor de 100! usando el siguiente código:

```r
n <- 100
result <- 1
for (i in 1:n)
result <- result * i
result
```

```
[1] 9.332622e+157
```

Ejemplo Fibonacci

La secuencia de Fibonacci es una secuencia famosa en matemáticas. Los dos primeros elementos se definen como $\{1, 1\}$. Los elementos subsecuentes se definen como la suma de los dos elementos anteriores.

Por ejemplo:

- El tercer elemento es 2 $(= 1 + 1)$

- El cuarto elemento es 3 $(= 1 + 2)$

- El quinto elemento es 5 $(= 2 + 3)$

- Y así sucesivamente.

Para obtener los primeros 12 números de Fibonacci en R, podemos usar:

```r
Fibonacci <- numeric(12) # vector con ceros para completar
Fibonacci[1] <- Fibonacci[2] <- 1
for (i in 3:12)
Fibonacci[i] <- Fibonacci[i - 2] + Fibonacci[i - 1]

print(Fibonacci)
```

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

Visualización de los números

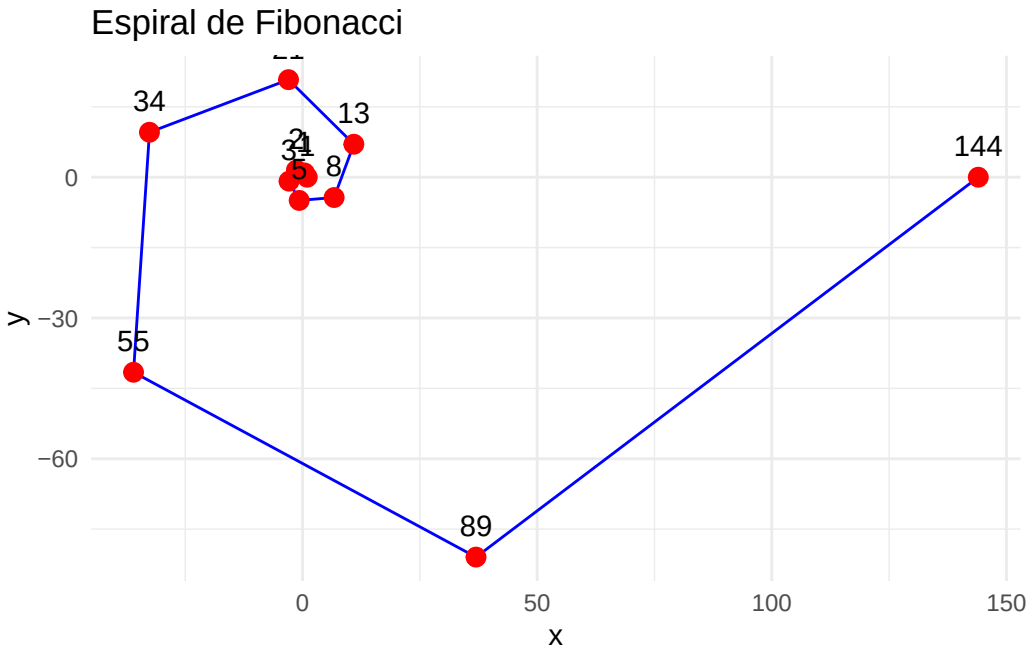
```r
# Crear ángulos para la espiral
angles <- seq(0, 4 * pi, length.out = length(Fibonacci))

# Graficar en coordenadas polares
library(ggplot2)
```

```r
# Crear un data frame
df <- data.frame(
  angle = angles,
  radius = Fibonacci,
  label = Fibonacci
)

# Convertir a coordenadas cartesianas para ggplot
df$x <- df$radius * cos(df$angle)
df$y <- df$radius * sin(df$angle)

# Graficar
ggplot(df, aes(x, y)) +
  geom_path(color = "blue") +
  geom_point(size = 3, color = "red") +
  geom_text(aes(label = label), vjust = -1, size = 4) +
  coord_equal() +
  theme_minimal() +
  ggtitle("Espiral de Fibonacci")
```



Uso de `if`

if (condición){ comandos cuando es VERDADERO}else{comandos cuando es FALSO}

```r
edad <- 20

if (edad >= 18) {
  print("Eres mayor de edad")
} else {
  print("Eres menor de edad")
}
```

```
[1] "Eres mayor de edad"
```

Creación de una función condicionada

```r
x1 <- rnorm(30, 1,2)
x2 <- 2 + 10*x1 + rnorm(30, 0,10)

# pequeña función
corplot <- function(x, y, plotit) {
if (plotit == TRUE) plot(x, y)
cor(x, y)
}

corplot(x1, x2,FALSE)
```

```
[1] 0.9170117
```

Uso de `while`

establecer x en 1 mientras (x <= 5): mostrar x aumentar x en 1

```r
x <- 1

while (x <= 5) {
  print(x)
  x <- x + 1
}
```

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

¿Cuantos lanzamientos de una moneda requiero realizar para obtener 10 caras ?

```r
# Inicializamos variables
caras <- 0
lanzamientos <- 0

# Bucle while: se repite hasta obtener 10 caras
while (caras < 10) {
  lanzamiento <- sample(c("cara", "cruz"), 1)
  lanzamientos <- lanzamientos + 1
  if (lanzamiento == "cara") {
    caras <- caras + 1
  }
}

# Resultado
cat("Se necesitaron", lanzamientos, "lanzamientos para obtener 10 caras.\n")
```

```
Se necesitaron 14 lanzamientos para obtener 10 caras.
```

Uso de `repeat` and `break`

repetir: mostrar x aumentar x en 1 hasta que (x > 5)

```r
x <- runif(1,0,7)

repeat {
  print(x)
  x <- x + 1
  if (x > 14) break
}
```

```
[1] 4.849615
[1] 5.849615
[1] 6.849615
[1] 7.849615
[1] 8.849615
[1] 9.849615
[1] 10.84961
[1] 11.84961
```

```
[1] 12.84961
[1] 13.84961
```

En este ejemplo se utiliza un bucle `repeat` junto con `break` para recorrer el conjunto de datos `mtcars` y encontrar el primer automóvil cuyo rendimiento en millas por galón (`mpg`) sea superior a 30. El bucle se ejecuta indefinidamente hasta que se cumple la condición deseada, momento en el cual se imprime el nombre del auto y su valor de `mpg`, y se interrumpe la ejecución con `break`. Este enfoque es útil cuando se desea identificar rápidamente el primer caso que cumple un criterio específico dentro de un conjunto de datos.

```r
# Cargar el dataset
data(mtcars)

# Convertir los nombres de fila en una columna para poder accederlos
mtcars$car <- rownames(mtcars)

# Inicializar índice
i <- 1

repeat {
  if (mtcars$mpg[i] > 30) {
    cat("El primer auto con mpg > 30 es:", mtcars$car[i], "con", mtcars$mpg[i], "mpg.\n")
    break
  }
  i <- i + 1
}
```

```
El primer auto con mpg > 30 es: Fiat 128 con 32.4 mpg.
```

Aplicación de loops al analisis de datos

En análisis de datos, es común encontrarse con valores faltantes (`NA`) que deben ser tratados antes de realizar cualquier análisis estadístico. Este ejemplo utiliza el dataset `airquality`, incluido en R, que contiene mediciones diarias de calidad del aire en Nueva York. El objetivo es demostrar cómo usar los bucles `for`, `while`, `repeat` y `break` para:

1. **Detectar columnas con datos faltantes**.
2. **Imputar los valores faltantes con la media de cada variable**.
3. **Verificar que no queden `NA`**.
4. **Generar un resumen estadístico del dataset limpio**.

Este caso nos permitirá practicar estructuras de control en R dentro de un contexto realista y útil para la limpieza de datos.

```
# Cargar dataset
data("airquality")
```

```
str(airquality)
ncol(airquality)
nrow(airquality)
nombres <- names(airquality)

airquality[nombres[1]]
airquality["Ozone"]
airquality[["Ozone"]]
airquality[[1]]

if(FALSE){print("Hello world")}

is.na(airquality[["Ozone"]])
any(is.na(airquality[["Ozone"]]))

# La matriz completa
any(is.na(airquality))

# Operadores Logicos
if(4>10 && 4>100) {print("Correcto")}
if(4<10 && 4<100) {print("Correcto")}
```

```
# Completar
```

```
cat("Columnas con valores NA:\n")
for (col in names(airquality)) {
  if (any(is.na(airquality[[col]]))) {
    cat("-", col, "\n")
  }
}
```

```
# 1. Identificar columnas con NA usando un bucle for
```

```
# 2. Imputar valores NA con la media de la columna usando while
```

```
col_index <- 1
while (col_index <= ncol(airquality)) {
  columna <- airquality[[col_index]]
  if (is.numeric(columna) && any(is.na(columna))) {
    media <- mean(columna, na.rm = TRUE)
    columna[is.na(columna)] <- media
    airquality[[col_index]] <- columna
  }
  col_index <- col_index + 1
}
```

```
# 2. Imputar valores NA con la media de la columna usando while
```

```
repeat {
  if (any(is.na(airquality))) {
    cat("Aún hay valores NA.\n")
    break
  } else {
    cat("Todos los valores NA han sido imputados correctamente.\n")
    break
  }
}
```

```
# 3. Verificar que no queden NA usando repeat y break
```

```
# 4. Resumen estadístico del dataset limpio
cat("\nResumen estadístico del dataset limpio:\n")
```

```
Resumen estadístico del dataset limpio:
```

```
print(summary(airquality))
```

```
     Ozone           Solar.R          Wind             Temp
 Min.   :  1.00   Min.   :  7.0   Min.   : 1.700   Min.   :56.00
 1st Qu.: 18.00   1st Qu.:115.8   1st Qu.: 7.400   1st Qu.:72.00
 Median : 31.50   Median :205.0   Median : 9.700   Median :79.00
 Mean   : 42.13   Mean   :185.9   Mean   : 9.958   Mean   :77.88
 3rd Qu.: 63.25   3rd Qu.:258.8   3rd Qu.:11.500   3rd Qu.:85.00
 Max.   :168.00   Max.   :334.0   Max.   :20.700   Max.   :97.00
```

```
 NA's   :37       NA's   :7
     Month            Day
 Min.   :5.000   Min.   : 1.0
 1st Qu.:6.000   1st Qu.: 8.0
 Median :7.000   Median :16.0
 Mean   :6.993   Mean   :15.8
 3rd Qu.:8.000   3rd Qu.:23.0
 Max.   :9.000   Max.   :31.0
```

Reproduciendo operaciones con functiones `function(arg)`

¿Qué es una función en R?

Una **función** en R es un bloque de código que realiza una tarea específica. Sirve para **organizar** y **reutilizar** código de forma más clara y eficiente.

¿Cómo se define una función?

Una función se define con la palabra clave `function`, seguida de paréntesis con los **argumentos** (si los hay), y luego un bloque de código entre llaves `{ }`.

Definir una función

```
definir función con nombre:
    (opcionalmente con argumentos)
    {
        hacer algo con esos argumentos
        devolver un resultado
    }
```

Ejemplo de función

```
corplot_v2 <- function(x, y, plotit = TRUE, method = "pearson", main = NULL, xlab = NULL, yla
  # Validaciones
  if (!is.numeric(x) || !is.numeric(y)) stop("Ambos vectores deben ser numéricos.")
  if (length(x) != length(y)) stop("Los vectores deben tener la misma longitud.")
  if (!method %in% c("pearson", "spearman", "kendall")) stop("Método no válido.")
```

```
  # Gráfico opcional
  if (plotit) {
    plot(x, y, main = main, xlab = xlab, ylab = ylab)
  }

  # Calcular correlación
  return(cor(x, y, method = method))
}
```

```
#Completar
```

```
#corplot_v2(x1,x2,TRUE)
```

Output con multiples objetos

```
library(ggplot2)

corplot_multi <- function(x, y, alpha = 0.05) {
  if (!is.numeric(x) || !is.numeric(y)) stop("Ambos vectores deben ser numéricos.")
  if (length(x) != length(y)) stop("Los vectores deben tener la misma longitud.")

  # Calcular correlación y prueba
  test <- cor.test(x, y)

  # Crear gráfico con ggplot2
  df <- data.frame(x = x, y = y)
  p <- ggplot(df, aes(x, y)) +
    geom_point() +
    geom_smooth(method = "lm", se = FALSE, col = "blue") +
    ggtitle("Gráfico de dispersión con línea de tendencia")

  # Salida como lista con descripciones
  return(list(
    correlacion = test$estimate,
    p_valor = test$p.value,
    significativa = test$p.value < alpha,
    grafico = p
  ))
}
```

```
library(ggplot2)
#Completar
```

```
# res <- corplot_multi(mtcars$mpg, mtcars$hp)
# print(res$correlacion)
# print(res$significativa)
# print(res$grafico)
```

```
#corplot_multi(x1,x2)$p
```

Output en clase `S3`

En R, los **objetos** pueden pertenecer a diferentes **clases**, que determinan cómo se comportan y cómo se interpretan por funciones genéricas como `print()`, `summary()` o `plot()`. Estas clases permiten organizar y estructurar los datos de forma coherente. Una de las formas más comunes y flexibles de definir clases en R es mediante el sistema **S3**, que es simple, dinámico y ampliamente utilizado.

El sistema **S3** no requiere una definición formal de clases. En cambio, se basa en la asignación de un atributo de clase a un objeto y en la creación de funciones específicas para esa clase. Por ejemplo, si se crea un objeto de clase `"corplot_result"`, se puede definir una función `print.corplot_result()` que se ejecutará automáticamente cuando se use `print()` sobre ese objeto. Esto permite personalizar el comportamiento de funciones genéricas según el tipo de objeto.

```
# Crear una función que devuelva un objeto con clase S3
corplot_s3 <- function(x, y) {
  resultado <- list(
    correlacion = cor(x, y),
    resumen = summary(lm(y ~ x))
  )
  class(resultado) <- "corplot_result"  # Asignar clase S3
  return(resultado)
}

# Definir un método print específico para la clase
print.corplot_result <- function(obj) {
  cat("Correlación:", obj$correlacion, "\n")
  cat("Resumen del modelo lineal:\n")
  print(obj$resumen)
}
```

```r
# Crear una función que devuelva un objeto con clase S3
```

```r
#res <- corplot_s3(mtcars$mpg, mtcars$hp)
#print(res)  # Llama automáticamente a print.corplot_result()
```

SESIÓN 2

Uso del operador "pipeline" %>%

El operador pipe %>% en R El operador %>% proviene del paquete `magrittr` y se usa para **encadenar funciones** de forma más legible. En lugar de escribir funciones anidadas como:

Podemos escribirlo de forma más clara y natural:

x %>% h() %>% g() %>% f()

Esto mejora la **lectura del código**, ya que se sigue el flujo de izquierda a derecha, como si se estuviera diciendo: "toma `x`, aplícale `h`, luego `g`, y finalmente `f`".

```r
###  Ejemplo en R
library(magrittr)
resultado <- 5 %>% sqrt() %>% round(2)
```

Este código toma el número `5`, le aplica la raíz cuadrada (`sqrt()`), y luego redondea el resultado a 2 decimales (`round(2)`).

```
Pseudocódigo para `%>%`
tomar x pasar x a función1 pasar resultado a función2 pasar resultado a función3 guardar resu
```

¿Por qué usar %>%?

- Hace que el código sea más **legible** - Permite escribir operaciones en **secuencia lógica** - Es muy útil cuando se trabaja con **manipulación de datos**, especialmente con `dplyr` y `tidyverse`

Uso de la función `replicate()`

Buenas prácticas en programamción