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

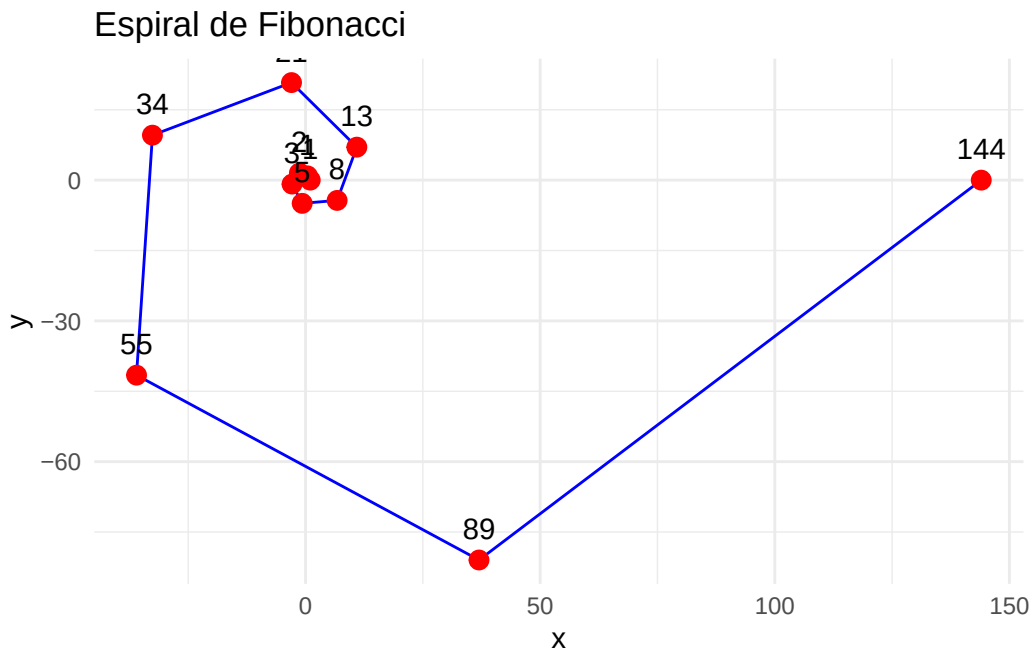
```r
# Crear ángulos para la espiral
angles <- seq(0, 4 * pi, length.out = length(Fibonacci))

# 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}

```
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

```
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] 0.9420322
```

Uso de `while`

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

```
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 22 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] 1.461204
[1] 2.461204
[1] 3.461204
[1] 4.461204
[1] 5.461204
[1] 6.461204
[1] 7.461204
[1] 8.461204
[1] 9.461204
[1] 10.4612
[1] 11.4612
[1] 12.4612
[1] 13.4612
```

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.

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

```r
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")}
```

```r
# 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  
```

SESIÓN 2

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`

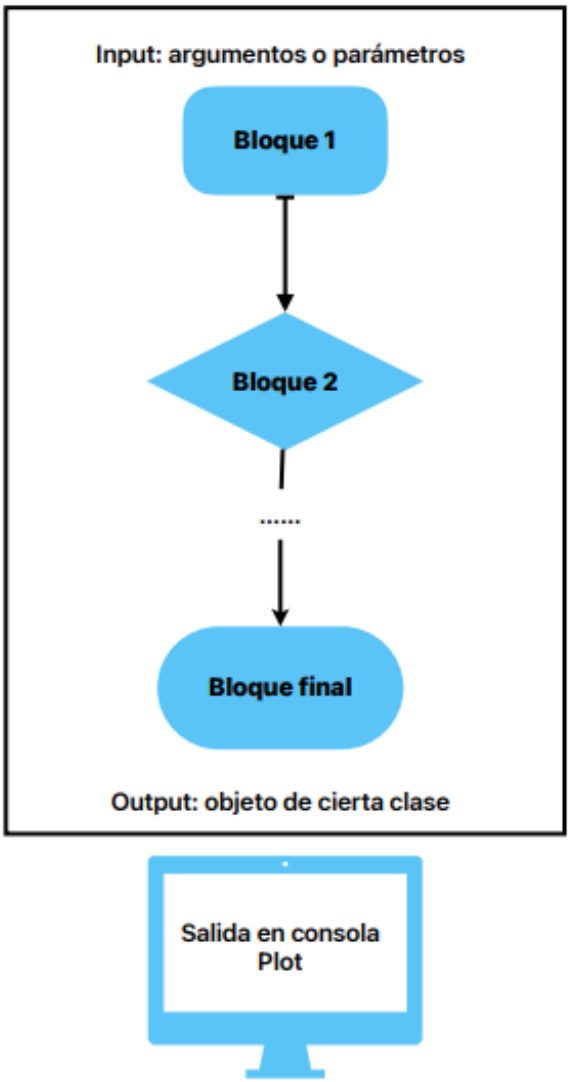


Figure 1: ¿Que hace una función?

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.

La siguiente función toma un dataset con NAs y lo convierte en otro dataframe sin NA usando inputación simple.

```
# ------------------------------------------------------------------------
# Paso: definimos la funcion, argumentos y bloques

imputar_na_num <- function(data, metodo = "media") {
  # data: El dataframe de entrada.
  # metodo: "media" (por defecto) o "mediana" para la imputación.

  # 1. Validación de entradas

  # 2. Copia del dataframe para no modificar el original

  # 3. Imputación de NAs en columnas numéricas

  # 4. Preparar la salida con clase S3

  # 5. Asignamos una clase a la lista. Esto permite un 'print' personalizado.
}
```

```
# Paso: definimos la funcion, argumentos y bloques
```

```
# ------------------------------------------------------------------------
# Paso 1: Validación de entradas

if (!is.data.frame(data)) {
  stop("Error: 'data' debe ser un dataframe.")
}
if (!metodo %in% c("media", "mediana")) {
  stop("Error: 'metodo' debe ser 'media' o 'mediana'.")
}
```

```
# Paso 1: Validación de entradas
```

```r
# ------------------------------------------------------------------------
# Paso 2: Copia del dataframe para no modificar el original

df_imputado <- data
columnas_imputadas <- character(0) # Para registrar qué columnas se modifican
```


```r
# ------------------------------------------------------------------------
# Paso 2: Copia del dataframe para no modificar el original
```


```r
# ------------------------------------------------------------------------------
# Paso 3: Imputación de NAs en columnas numéricas
for (col_name in names(df_imputado)) {
  columna <- df_imputado[[col_name]]

  if (is.numeric(columna) && any(is.na(columna))) {
    # Calcula el valor de imputación (media o mediana)
    valor_imputacion <- switch(metodo,
                               "media" = mean(columna, na.rm = TRUE),
                               "mediana" = median(columna, na.rm = TRUE))

    # Realiza la imputación
    df_imputado[is.na(df_imputado[[col_name]]), col_name] <- valor_imputacion
    columnas_imputadas <- c(columnas_imputadas, col_name)
  }
}
```


```r
# ------------------------------------------------------------------------------
# Paso 3: Imputación de NAs en columnas numéricas
```


```r
# ------------------------------------------------------------------------------
# Paso 4: Preparar la salida como lista
# La función devuelve una lista con el dataframe imputado y un resumen breve.
resultado <- list(
  data_imputed = df_imputado,
  method_used = metodo,
  columns_affected = columnas_imputadas
)
```


```r
# ------------------------------------------------------------------------------
# Paso 4: Preparar la salida como lista
```

```
# Paso 5: Asignamos una clase a la lista. Esto permite un 'print' personalizado.
class(resultado) <- "imputacion_data"

return(resultado)
```

```
# Paso 5: Asignamos una clase a la lista. Esto permite un 'print' personalizado.
```

La función completa resultante es la siguiente

```
imputar_na_num <- function(data, metodo = "media") {
  # data: El dataframe de entrada.
  # metodo: "media" (por defecto) o "mediana" para la imputación.

  # 1. Validación de entradas
  if (!is.data.frame(data)) {
    stop("Error: 'data' debe ser un dataframe.")
  }
  if (!metodo %in% c("media", "mediana")) {
    stop("Error: 'metodo' debe ser 'media' o 'mediana'.")
  }

  # 2. Copia del dataframe para no modificar el original
  df_imputado <- data
  columnas_imputadas <- character(0) # Para registrar qué columnas se modifican

  # 3. Imputación de NAs en columnas numéricas
  for (col_name in names(df_imputado)) {
    columna <- df_imputado[[col_name]]

    if (is.numeric(columna) && any(is.na(columna))) {
      # Calcula el valor de imputación (media o mediana)
      valor_imputacion <- switch(metodo,
                                 "media" = mean(columna, na.rm = TRUE),
                                 "mediana" = median(columna, na.rm = TRUE))

      # Realiza la imputación
      df_imputado[is.na(df_imputado[[col_name]]), col_name] <- valor_imputacion
      columnas_imputadas <- c(columnas_imputadas, col_name)
    }
  }

  # 4. Preparar la salida como lista
```

```
  # La función devuelve una lista con el dataframe imputado y un resumen breve.
  resultado <- list(
    data_imputed = df_imputado,
    method_used = metodo,
    columns_affected = columnas_imputadas
  )

  # 5. Asignamos una clase a la lista. Esto permite un 'print' personalizado.
  class(resultado) <- "imputacion_data"

  return(resultado)
}
```

```
# Test
df <- imputar_na_num(airquality)
any(is.na(df$data_imputed))
```

```
[1] FALSE
```

Con el objetivo personalizar el ouput de la función `imputar_na_num`, se agregan atributos especiales a esta (clase S3)

```
# Clase S3: proveemos una opcion de impresión más informativa al objeto "imputacion_data"

print.imputacion_data <- function(x, ...) {

  # Paso 6: encabezado de la salida

  # Paso 7: identificamos las columnas afectadas

  # Paso 8: preparamos el texto de salida

}
```

```
# Clase S3: proveemos una opcion de impresión más informativa al objeto "imputacion_data"


# Paso 6: encabezado de la salida
  cat("--- Resultado de la Imputación de NAs ---\n")
  cat("Método de imputación usado:", x$method_used, "\n")
```

```
# Paso 6: encabezado de la salida
```

```
# Paso 7: identificamos las columnas afectadas
  if (length(x$columns_affected) > 0) {
    cat("Columnas numéricas afectadas por la imputación:\n")
    for (col in x$columns_affected) {
      cat("-", col, "\n")
    }
  } else {
    cat("No se encontraron NAs en columnas numéricas o no se realizaron imputaciones.\n")
  }
```

```
# Paso 7: identificamos las columnas afectadas
```

```
  # Paso 8: preparamos el texto de salida
  cat("\nPrimeras filas del dataframe imputado:\n")
  print(head(x$data_imputed))
  cat("\nPara acceder al dataframe completo, usa: 'nombre_tu_resultado$data_imputed'.\n")
```

```
  # Paso 8: preparamos el texto de salida
```

El código completo es:

```
# Clase S3: proveemos una opcion de impresión más informativa al objeto "imputacion_data"

print.imputacion_data <- function(x, ...) {

  # Paso 6: encabezado de la salida
  cat("--- Resultado de la Imputación de NAs ---\n")
  cat("Método de imputación usado:", x$method_used, "\n")

  # Paso 7: identificamos las columnas afectadas
  if (length(x$columns_affected) > 0) {
    cat("Columnas numéricas afectadas por la imputación:\n")
    for (col in x$columns_affected) {
      cat("-", col, "\n")
    }
  } else {
```

```
    cat("No se encontraron NAs en columnas numéricas o no se realizaron imputaciones.\n")
  }

  # Paso 8: preparamos el texto de salida
  cat("\nPrimeras filas del dataframe imputado:\n")
  print(head(x$data_imputed))
  cat("\nPara acceder al dataframe completo, usa: 'nombre_tu_resultado$data_imputed'.\n")
}
```

```
# test
df <- imputar_na_num(airquality)
print(df)
```

```
--- Resultado de la Imputación de NAs ---
Método de imputación usado: media
Columnas numéricas afectadas por la imputación:
- Ozone
- Solar.R

Primeras filas del dataframe imputado:
     Ozone  Solar.R Wind Temp Month Day
1 41.00000 190.0000  7.4   67     5   1
2 36.00000 118.0000  8.0   72     5   2
3 12.00000 149.0000 12.6   74     5   3
4 18.00000 313.0000 11.5   62     5   4
5 42.12931 185.9315 14.3   56     5   5
6 28.00000 185.9315 14.9   66     5   6

Para acceder al dataframe completo, usa: 'nombre_tu_resultado$data_imputed'.
```

```
summary(df)
```

```
                 Length Class      Mode
data_imputed     6      data.frame list
method_used      1      -none-     character
columns_affected 2      -none-     character
```

Depuración (debugging) y mantenimiento

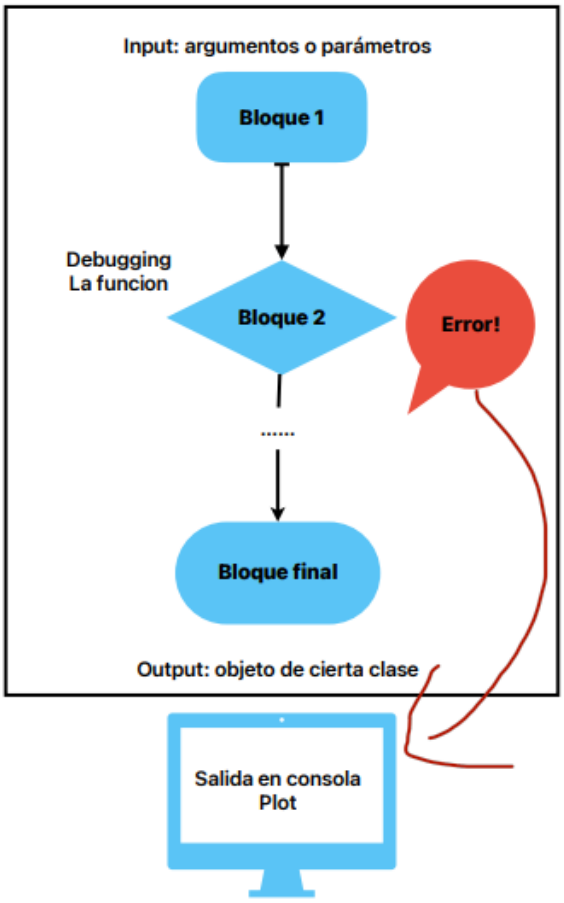


Figure 2: ¿Que hace una función?

Los errores en los programas se llaman **bugs**. El proceso de eliminarlos se llama **depuración**. Aunque el objetivo es escribir programas sin errores, a veces cometemos equivocaciones.

Cinco pasos para depurar un programa:

1. **Reconocer que existe un error.**
2. **Hacer que el error sea reproducible.**
3. **Identificar la causa del error.**
4. **Corregir el error y probar.**
5. **Buscar errores similares.**

Reconociendo que existe un error

A veces es evidente: el programa no funciona. Pero en otros casos, el programa parece funcionar, aunque da resultados incorrectos o falla con ciertos datos de entrada. Este tipo de errores son más difíciles de detectar.

Estrategia recomendada:
Dividir el programa en partes simples, como se sugiere en secciones anteriores del texto.

Buenas Prácticas para Escribir y Probar Funciones

Al desarrollar funciones en programación, es importante seguir estas recomendaciones:

- **Documentar entradas y salidas**: Especificar claramente qué datos recibe y qué devuelve la función.
- **Probar supuestos sobre las entradas**: Verificar que los datos de entrada cumplen con lo esperado.
- **Considerar casos límite**: Asegurarse de que la función maneje correctamente situaciones extremas o inusuales.
- **Escribir dos versiones de la función**:
  - Una versión simple para verificar la **corrección**.
  - Una versión optimizada para mejorar el **rendimiento**.

Identificar la Causa del Error

Una vez que se confirma la existencia de un bug, el siguiente paso es **identificar su causa**.

Pasos recomendados:

- **Leer los mensajes de error**: Intenta comprenderlos lo mejor posible.
- **Buscar ayuda**:
  - Si el mensaje no es claro, intenta buscarlo en línea.
  - También puedes pedir ayuda a alguien con más experiencia.
- **Usar `traceback()` en R**:
  - Esta función muestra la pila de funciones activas en el momento del error.
  - Es útil para rastrear dónde ocurrió el problema.

Ejemplos

```
# Var: función de varinza
# sqrt(var)
```

```
#library(randompackage)
```

```
#if(x==NA)print("NA")
```

```
data(iris)
#print.imputacion_data(iris$Species)
```

La función `traceback()` en R muestra la secuencia de llamadas a funciones (la pila de ejecución) que llevó al error, ayudando a identificar dónde ocurrió el problema en el código.

```
# Usar en la consola
# traceback()
```

Cómo Funciona `debug()` en R para Depuración

La función **`debug()`** en R es una herramienta esencial para inspeccionar el código de una función paso a paso y entender su comportamiento o encontrar errores. Cuando aplicas `debug()` a una función, R entra en un modo interactivo especial (**modo de depuración**) la próxima vez que esa función es llamada.

Mecanismo:

1. **Activación:** Usas `debug(nombre_de_tu_funcion)`.
2. **Ejecución:** La próxima vez que llamas a `nombre_de_tu_funcion`, R no la ejecuta normalmente. En su lugar, detiene la ejecución al principio de la función.
3. **Modo Interactivo:** Tu consola de R cambiará a un *prompt* especial, usualmente `Browse[1]>`. En este modo, puedes ejecutar comandos de depuración para controlar el flujo del código y examinar el estado de las variables.
4. **Desactivación:** Para que la función vuelva a ejecutarse normalmente, usas `undebug(nombre_de_tu_funcion)`.

Comandos Útiles en Modo de Depuración (`Browse[1]>`)

Cuando estás en el *prompt* `Browse[1]>`, puedes usar los siguientes comandos para navegar por tu código:

- `n` (Next): Ejecuta la **siguiente línea** de código. Si la siguiente línea es una llamada a otra función, no entrará en esa función; simplemente ejecutará la función como un todo.
- `s` (Step Into): Ejecuta la siguiente línea. Si la siguiente línea es una **llamada a otra función**, `s` te permite **entrar** en esa función para depurarla también, paso a paso.
- `f` (Finish): Ejecuta el **resto del bucle actual o la función actual** y sale de ella, volviendo al nivel de depuración superior o a la ejecución normal.
- `c` (Continue): Ejecuta el código **hasta el final** de la función o hasta el próximo punto de interrupción (si lo hubiera).
- `Q` (Quit): **Sale inmediatamente del modo de depuración** y detiene la ejecución de la función.
- `where`: Muestra la **pila de llamadas** (`call stack`), indicando en qué funciones anidadas te encuentras actualmente. Es útil para saber dónde estás en el flujo de ejecución.
- `ls()`: Lista las variables y objetos disponibles en el **entorno actual** (dentro de la función o bucle).
- `nombre_de_variable`: Simplemente escribe el nombre de cualquier variable que esté en el entorno actual para **ver su valor**.

Otras Opciones de Depuración

Además de `debug()`, R ofrece otras herramientas para depurar. `browser()` permite insertar pausas específicas en el código. `debugonce()` depura solo la siguiente ejecución de la función. Para usuarios de RStudio, los **puntos de interrupción gráficos** son la forma más visual y sencilla de depurar paso a paso, sin modificar el código.

Ejemplo

```
#
#debug(imputar_na_num)
#
#imputar_na_num(iris)
```

```
#debug(imputar_na_num)

#imputar_na_num(iris$Species)
```

```
undebug(imputar_na_num)
```

```
Warning in undebug(imputar_na_num): argument is not being debugged
```

Veamos un ejemplo solo con variables cualitativas:

```
# imputar_na_num(c(10, 20, 30, NA, 50))
```