Capitulo 7

Programando en R

Ejemplo Fibonacci

```
Fib1 <- 1
Fib2 <- 1
Fib3 <- Fib2 + Fib1
Fib4 <- Fib3 + Fib2
Fib5 <- Fib4 + Fib3
```

Fuerza bruta

Dado que R proporciona capacidades aritméticas básicas, las más simples, y las menos útiles, el enfoque para calcular los números de Fibonacci sería utilizar R para implementar los cálculos de la fuerza bruta. Por ejemplo, la siguiente secuencia de pasos calcula los primeros cinco números de Fibonacci:

```
Fib <- vector("numeric", length = 5)
Fib[1] <- 1
Fib[2] <- 1
Fib[3] <- Fib[2] + Fib[1]
Fib[4] <- Fib[3] + Fib[2]
Fib[5] <- Fib[4] + Fib[3]
```

Vale la pena enfatizar que este ejemplo no es más un programa real que el anterior, aunque la introducción del vector Fib es un paso en la dirección correcta, formando la base para el siguiente ejemplo.

Usando bucles: un primer programa real

Las estructuras de control representan un componente importante de cualquier lenguaje de programación, uno de los más simples de Estas estructuras de control son el bucle for utilizado aquí:

```
n <- 30
Fib <- vector("numeric", length = n)
Fib[1] <- 1
Fib[2] <- 1
for (i in 3:n){
Fib[i] <- Fib[i-1] + Fib[i-2]
}
```

Una manera alternativa con ciclo for

```
Fib <- vector("numeric", length = 30)
Fib[1] <- 1
Fib[2] <- 1
for (i in 3:30){
Fib[i] <- Fib[i-1] + Fib[i-2]
}
```

Un truco de inicializacion

```
n <- 30
Fib <- rep(1, n)
for (i in 3:n){
Fib[i] <- Fib[i-1] + Fib[i-2]
}
```

La funcion Fibonacci personalizada

```
Fibonacci <- function(n){
Fib <- rep(1, n)
for (i in 3:n){
Fib[i] <- Fib[i-1] + Fib[i-2]
}
return(Fib)
}
```

Estructura de la funcion:

FunctionName <- function(Argument1, . . . , ArgumentN){ (Body: executable code goes here) return(Result) }

Creando tus propias Funciones

En un nivel superior, la tarea de escribir su propia función consta de estos pasos:

1. Decide qué quieres que haga tu función:

a. Decida el nombre de su función;si.
b. Decida con qué argumentos debe llamarse el programa;
c. Decida qué cálculos deben ocurrir en el cuerpo de la función;
d. Decida qué objeto de datos debe devolver la función.

2. Cree un archivo externo que contendrá su función, por ejemplo:

a. Use la función file.create para crear el archivo; b.Use la función file.edit para permitirle editarlo;
b. Cree el texto de la función y guarde el resultado.

3. Use la fuente para llevar su función a su sesión interactiva de R;

4. Pruebe su programa en algunos ejemplos con resultados conocidos;

5. Decide si tu programa está funcionando:

a. Si es así, celebre y use su programa;
b. Comience el proceso de depuración.

Diseñando tu función

Es una buena idea darle a su función un nombre que le permita a usted, y a cualquier otra persona con la que elija compartirlo, ver lo más claramente posible lo que hace la función solo por su nombre.

Este nombre se convertirá en la palabra clave en la primera línea de código que define la función, por lo que debe ser una palabra, aunque las concatenaciones de palabras “CamelCase” son aceptables, a menudo permitiendo nombres más informativos (por ejemplo, FindOutliers).

Además, es una buena práctica usar este nombre de función como el nombre del archivo que crea para mantener la función, el paso que se analiza a continuación; por ejemplo, si está creando la función llamada FindOutliers, el nombre del archivo debe ser FindOutliers.R, que le indica inmediatamente al buscar en el directorio, primero, que este archivo contiene el código fuente R y, en segundo lugar, lo que la funcion contiene.

Creando tu funcion archivo

```
#file.create("ComputeIQD.R")
#file.edit("ComputeIQD.R")
```

El nombre de este programa será ComputeIQD, que se guardará en el archivo ComputeIQD.R. La tabla de IPO para este ejemplo se ve así: I: El único argumento de entrada es el vector numérico x. P: Procesamiento realizado por el cuerpo de la función: 1. Calcule los cuartiles superior e inferior del vector x. 2. Calcule el IQD

como la diferencia entre estos cuartiles. O: Salida: devuelve el valor de IQD. Como se señaló anteriormente, creamos y editamos el archivo fuente con estos comandos:

```
#file.create("ComputeIQD.R")
#file.edit("ComputeIQD.R")
```

Estructuras de control

Al igual que todos los lenguajes de programación, R admite varias estructuras de control, pero algunas se usan con más frecuencia que otras. Las siguientes discusiones se centran en cuáles son probablemente las dos estructuras de control más comunes para bucles y estructuras if-else y luego introduce la función vectorial ifelse que puede ser extremadamente útil y probablemente no sea tan conocida como debería ser.

Estructuras de bucle Si bien R generalmente desalienta el uso de bucles (consulte la Sección 7.2.4 para una discusión de este punto), el lenguaje admite varios tipos de bucles. El único de estos discutido aquí es el bucle for que vimos en la secuencia de Fibonacci, la estructura básica de este bucle es la siguiente:

for (Element in Set){ Computations for each Element value }

```
n <- 30
Fib <- vector("numeric", length = n)
Fib[1] <- 1
Fib[2] <- 1
for (i in 3:n){
Fib[i] <- Fib[i-1] + Fib[i-2]
}
```

Secuencias If-else

Además de los bucles, las otras estructuras de control que surgen con mayor frecuencia en los programas R son aquellas basadas en declaraciones if simples, pares if / else o cadenas de pares if / else. La estructura general de la declaración if individual es:

if (Condcion){ Se ejecuta lo que esta dentro de las llaves }

Aquí, la condición es cualquier declaración lógica que se evalúe como VERDADERO o FALSO: si es VERDADERO, se ejecuta el bloque de código sangrado; de lo contrario, este código se omite.

El par if / else es una estructura de control más flexible que permite que se ejecute un conjunto de código si la Condición es VERDADERA y que otro conjunto se ejecuta si es FALSO.

El formato de esta estructura es:

if (Condicion){ Primera ejecucion dentro de las llaves de la condicion } else { Segunda ejecucion dentro de las llaves } La funcion ifelse

La función ifelse se llama con una prueba de vector lógico y dos vectores de posibles valores de retorno, sí y no, cada uno de la misma longitud que la prueba. Para cada elemento de prueba que es VERDADERO, ifelse devuelve el elemento correspondiente de sí, y para cada elemento de prueba que es FALSO, devuelve el elemento correspondiente de no.

Esta función es útil para tareas como reemplazar valores perdidos por cero:

```
set.seed(3)
x <- rnorm(10)
x
```

```
##  [1] -0.96193342 -0.29252572  0.25878822 -1.15213189  0.19578283
##  [6]  0.03012394  0.08541773  1.11661021 -1.21885742  1.26736872
```

```
y <- sqrt(x)
```

```
## Warning in sqrt(x): Se han producido NaNs
```

```
y
```

```
##  [1]       NaN       NaN 0.5087123       NaN 0.4424735 0.1735625 0.2922631
##  [8] 1.0566978       NaN 1.1257747
```

```
z <- ifelse(is.na(y), 0, y)
z
```

```
##  [1] 0.0000000 0.0000000 0.5087123 0.0000000 0.4424735 0.1735625 0.2922631
##  [8] 1.0566978 0.0000000 1.1257747
```

Valores Perdidos if-else

En las funciones if y ifelse se supone que una expresión lógica puede devolver solo dos valores, VERDADERO o FALSO.

En la lógica de tres valores, la posibilidad de que una o más de las variables involucrado en la expresión lógica puede tener un valor perdido o desconocido significa que el resultado de la expresión lógica puede faltar o ser desconocido.

```
x <- c(-1, 0, 1, 2, 3, NA, 4, 5, -6)
x > 0
```

```
## [1] FALSE FALSE  TRUE  TRUE  TRUE    NA  TRUE  TRUE FALSE
```

```
## [1] FALSE FALSE TRUE TRUE TRUE NA TRUE TRUE FALSE
y <- ifelse(x >= 0, paste(x, "non-negative"), paste(x, "negative"))
y
```

```
## [1] "-1 negative"    "0 non-negative" "1 non-negative" "2 non-negative"
## [5] "3 non-negative" NA               "4 non-negative" "5 non-negative"
## [9] "-6 negative"
```

```
## [1] "-1 negative" "0 non-negative" "1 non-negative" "2 non-negative"
## [5] "3 non-negative" NA "4 non-negative" "5 non-negative"
## [9] "-6 negative"
```

En la descripción de la función ifelse dada anteriormente, se supuso que la expresión lógica aquí x> 0 devolvería VERDADERO, lo que provocaría que la primera de las dos expresiones siguientes se devuelva como resultado, o FALSO, causando la segunda expresión ser devuelto.

En este ejemplo, cuando la expresión x> 0 devuelve el valor faltante NA, la función ifelse también devuelve este valor, si bien es difícil argumentar a favor de cualquier otro resultado como más razonable aquí, el resultado es inesperado si no consideramos la posibilidad de que x pueda tener valores faltantes, y en algunos casos esto puede conducir a errores de programa.

Alternativamente, en la estructura de control if-else discutida anteriormente, los valores faltantes en la expresión lógica en la instrucción if harán que la ejecución del programa termine con un error.

Como ejemplo específico, si no faltan valores de x, la siguiente secuencia if-else produciría un resultado similar al ejemplo ifelse que se acaba de describir, pero los valores de x perdidos hacen que el ciclo se detenga cuando la instrucción if encuentra un valor faltante:

```
#for (xEl in x){
#if (xEl >= 0){
#print(paste(xEl, "non-negative"))
#} else {
#print(paste(xEl, "negative"))
#}
#}
#[1]"-1 negative"
```

```
#[1]"0 non-negative"
#[1]"1 non-negative"
#[1]"2 non-negative"
#[1]"3 non-negative"
## Error in if (xEl >= 0) {:missing value where TRUE/FALSE needed
```

Null no es lo mismo que NA

Un punto extremadamente importante pero sutil es que los dos valores especiales de R NA y NULL no son lo mismo. Específicamente, el valor especial NA es el indicador estándar de R para los datos faltantes e influye en la lógica if-else en la forma que se acaba de describir. Por el contrario, el valor especial NULL que forma la base del "truco NULL".

```
zList <- list(a = "a", b = "B", c = NA, d = NULL, e = "eee")
zList
```

```
## $a
## [1] "a"
##
## $b
## [1] "B"
##
## $c
## [1] NA
##
## $d
## NULL
##
## $e
## [1] "eee"
```

```
a <- NA
length(a)
```

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

```
is.na(a)
```

```
## [1] TRUE
```

```
is.null(a)
```

```
## [1] FALSE
```

```
b <- NULL
is.null(b)
```

```
## [1] TRUE
```

```
length(b)
```

```
## [1] 0
```

```
is.na(b)
```

```
## logical(0)
```

```
#if (b > 0){
#print("b is positive")
#}
```

```
## Error in if (b > 0) {:argument is of length zero
c <- seq(0, 9, 1)
c
```

```
##  [1] 0 1 2 3 4 5 6 7 8 9
```

```
#c[6] <- NULL
## Error in c[6] <- NULL: replacement has length zero
c
```

```
##  [1] 0 1 2 3 4 5 6 7 8 9
```

El primer ejemplo, a, muestra que NA es un valor que puede asignarse a una variable, dando un objeto de longitud 1 que puede probarse con la función is.na, y para el cual la función is.null devuelve el valor FALSE , enfatizando que NA y NULL son diferentes.

El segundo ejemplo, b, muestra que NULL puede asignarse a una variable y la función is.null detecta este valor; Además, este valor tiene longitud cero (como la cadena de caracteres en blanco ") y provoca un error en expresiones lógicas como is.na (b) o b> 0. El tercer ejemplo, c, muestra que podemos asignar el valor NA a cualquier elemento de un vector existente, pero el intento de asignar el valor NULL falla, sin cambios en el vector original. zList$e <- NULL zList Finalmente, tenga en cuenta que el valor NULL surge con frecuencia y es particularmente útil al tratar con elementos de lista con nombre, como ilustra el siguiente ejemplo:

```
zList <- list(a = "a", b = "B", c = NA, d = NULL, e = "eee")
zList
```

```
## $a
## [1] "a"
##
## $b
## [1] "B"
##
## $c
## [1] NA
##
## $d
## NULL
##
## $e
## [1] "eee"
```

```
zList$e <- NULL
zList
```

```
## $a
## [1] "a"
##
## $b
## [1] "B"
##
## $c
## [1] NA
##
## $d
## NULL
```

Remplazando loops con funciones apply

Don't do this!

n <- length(x) z <- vector("numeric", length = n) for (i in 1:n){ z[i] <- x[i] + y[i] }

Los bucles son una estructura de control extremadamente importante, presente en todos los lenguajes de computadora, incluido R como acabamos de ver. Sin embargo, en R, los bucles explícitos son a menudo más lentos que otras construcciones de programas que nos permiten hacer lo mismo.

Un ejemplo específico son operaciones numéricas simples en vectores. Es decir, dados los vectores x e y, podemos obtener el vector cuyos elementos contienen las sumas de cada elemento de x e y directamente como x + y. En lenguajes que no admiten operaciones aritméticas de vectores, tendríamos que calcular la suma de vectores haciendo un bucle sobre los elementos individuales,

```
set.seed(3)
sapply(c(1, 2, 4, 8), rnorm)
```

```
## [[1]]
## [1] -0.9619334
##
## [[2]]
## [1] -0.2925257  0.2587882
##
## [[3]]
## [1] -1.15213189  0.19578283  0.03012394  0.08541773
##
## [[4]]
## [1]  1.1166102 -1.2188574  1.2673687 -0.7447816 -1.1312186 -0.7163585
## [7]  0.2526524  0.1520457
```

En particular, tenga en cuenta que cada elemento de la lista devuelto por sapply es un vector cuya longitud es igual al elemento correspondiente del vector con el que fue llamado. Finalmente, la función de aplicación se llama con una matriz o marco de datos, un parámetro MARGIN y una función que se aplicará a las filas (si MARGIN = 1) o columnas (si MARGIN = 2) de este objeto de datos de entrada.

Funciones genericas revisadas

En la breve discusión de la programación orientada a objetos. en R dado allí, se observó que R admite tres sistemas diferentes orientados a objetos, basados en objetos S3, objetos S4 y clases de referencia, y también se observó que el sistema de objetos S3 es el más simple y el que encontramos frecuentemente.

Asociados con objetos S3 hay métodos para funciones genéricas.

Al igual que plot, cuando se llama a esta función con un objeto que pertenece a una clase S3 específica, la función realiza las acciones apropiadas para esa clase, en el proceso de creación de sus propias clases S3 y sus métodos asociados, que consta de los siguientes pasos:

1. defina un nuevo nombre de clase para sus objetos S3.
2. use la función de clase para asignar objetos a esta clase.
3. cree un nuevo método para cualquier función genérica existente para esta clase, o cree una nueva función genérica con un método para esta clase.

```
library(MASS)
head(whiteside)
```

```
##    Insul Temp Gas
```

```
## 1 Before -0.8 7.2
## 2 Before -0.7 6.9
## 3 Before  0.4 6.4
## 4 Before  2.5 6.0
## 5 Before  2.9 5.8
## 6 Before  3.2 5.8
```

```
modelEx <- lm(Gas ~ Temp, data = whiteside)
class(modelEx)
```

```
## [1] "lm"
```

```
AnnotatedGaussianSample <- function(n, mean = 0, sd = 1, iseed = 33){
#
set.seed(iseed)
x <- rnorm(n, mean, sd)
ags <- list(n = n, mean = mean, sd = sd, seed = iseed, x = x)
class(ags) <- "AnnotatedGaussianSample"
return(ags)
}
y <- AnnotatedGaussianSample(100)
class(y)
```

```
## [1] "AnnotatedGaussianSample"
```

Las primeras ocho líneas aquí definen la función AnnotatedGaussianSample, que se llama con un argumento requerido y tres argumentos opcionales, y devuelve una lista de cinco elementos que proporciona los valores de estos argumentos y un vector de variables aleatorias gaussianas generadas a partir de ellos.Antes de devolver este objeto, la función le asigna la clase "AnnotatedGaussianSample", como se ve en la última línea de código aquí: la línea anterior llama a la función con n = 100, que devuelve el objeto y que tiene esta clase. Podemos obtener una vista más completa de la estructura de este objeto, incluida su clase, con la función str:

```
str(y)
```

```
## List of 5
##  $ n   : num 100
##  $ mean: num 0
##  $ sd  : num 1
##  $ seed: num 33
##  $ x   : num [1:100] -0.1359 -0.0408 1.0105 -0.1583 -2.1566 ...
##  - attr(*, "class")= chr "AnnotatedGaussianSample"
```

Agregando metodos a la funcion generica

Agregar un método a una función genérica existente es extremadamente fácil. Específicamente, si foo es una función genérica existente y ha definido una clase de objeto "new-Class", crear un nuevo método para foo solo requiere escribir una función llamada foo.newClass que haga lo que desea que haga foo cuando se le presente un objeto de la clase "newClass". Como ejemplo específico, la creación de la siguiente función agrega un método de resumen para los objetos de la clase "AnnotatedGaussianSample" creados por la función descrita anteriormente:

```
summary.AnnotatedGaussianSample <- function(x){
#
print(paste("Sample size:", x$n))
print(paste("Mean:", x$mean))
print(paste("Standard deviation:", x$sd))
print(paste("Random seed:", x$seed))
print("Sample quantiles:")
```

```
quantile(x$x)
}
```

Dado un objeto de la clase "AnnotatedGaussianSample", este método extrae los valores de los argumentos utilizados para configurar y llamar a la función rnorm, los muestra y luego calcula y muestra el resumen de cinco números de Tukey para la secuencia gaussiana aleatoria resultante. Para invocar este método, simplemente aplicamos la función genérica al objeto

```
summary(y)
```

```
## [1] "Sample size: 100"
## [1] "Mean: 0"
## [1] "Standard deviation: 1"
## [1] "Random seed: 33"
## [1] "Sample quantiles:"
```

```
##          0%         25%         50%         75%        100%
## -2.17699470 -0.58222699  0.01886743  0.70752861  2.73116619
```

A modo de comparación, supongamos que creamos exactamente la misma secuencia de números aleatorios gaussianos, pero directamente, sin llamar a la función Muestra Gaussiana Anotada, esto daría resultados diferentes:

```
set.seed(33)
x <- rnorm(n = 100, mean = 0, sd = 1)
class(x)
```

```
## [1] "numeric"
```

```
summary(x)
```

```
##     Min.  1st Qu.   Median     Mean  3rd Qu.     Max.
## -2.17699 -0.58223  0.01887  0.05949  0.70753  2.73117
```

```
methods(class = "AnnotatedGaussianSample")
```

```
## [1] summary
## see '?methods' for accessing help and source code
```

Creando nuevas funciones

```
foo <- function(x){UseMethod("foo")}
foo.AnnotatedGaussianSample <- function(x){
#
print("This is the foo method for AnnotatedGaussianSample S3 objects")
summary(x)
}
foo(y)
```

```
## [1] "This is the foo method for AnnotatedGaussianSample S3 objects"
## [1] "Sample size: 100"
## [1] "Mean: 0"
## [1] "Standard deviation: 1"
## [1] "Random seed: 33"
## [1] "Sample quantiles:"
```

```
##          0%         25%         50%         75%        100%
## -2.17699470 -0.58222699  0.01886743  0.70752861  2.73116619
```

La funcion ValidatiionRsquared

La función ValidationRsquared se propuso la medida de R al cuadrado de validación como una medida útil de la calidad de predicción para modelos predictivos de tipo arbitrario. Esta función es relativamente simple y relativamente estándar, ya que se llama con dos argumentos, ambos requeridos, y devuelve un resultado numérico

```
#ValidationRsquared
```

La funcion TVHsplit

Esta función se llama con cuatro argumentos, uno requerido y los otros tres opcionales con valores predeterminados especificados en la definición de la función. Específicamente, estos argumentos son:

1. el argumento requerido df, un marco de datos con N filas;
2. el argumento opcional dividido, un vector numérico de tres componentes;
3. las etiquetas de argumentos opcionales, un vector de caracteres de tres componentes;
4. el argumento entero opcional iseed. Esta función inicializa el sistema de muestreo aleatorio de R con el parámetro iseed y luego llama a la función de muestra a la muestra con reemplazo de los valores definidos por las etiquetas con las probabilidades definidas por división. El número de muestras extraídas corresponde al número de filas del marco de datos df, y el resultado devuelto es un vector de caracteres que asigna cada fila del marco de datos a uno de los subconjuntos especificados por la variable de etiquetas.

```
#TVHsplit
#function(df, split = c(0.5, 0.25, 0.25),
         #labels = c("T", "V", "H"), iseed = 397){
  #set.seed(iseed)
  #flags <- sample(labels, size = nrow(df), prob = split,replace = TRUE)
  #return(flags)
 #}
```

La funcion PredictedVsObservedPlot

Esta función utiliza todos los tipos de argumentos descritos en, incluidos los argumentos opcionales y obligatorios, un argumento opcional especificado utilizando el "truco NULL" y los argumentos con nombre opcionales pasados a la función de trazado básica de R a través de ". . ." argumento.

La función acepta los siguientes argumentos: 1. Observado es un vector numérico requerido de valores observados para ser trazados. 2. Predicted es un vector numérico requerido de predicciones del modelo de observado. 3. El argumento opcional xyLimits es NULL o un vector de dos componentes que especifica valores comunes para los argumentos xlim e ylim de la función de trazado, el valor predeterminado es NULL. 4.refLine es un argumento lógico opcional que especifica si se debe agregar una línea de referencia discontinua al diagrama. El valor por defecto es verdadero. 5.Los parámetros con nombre opcionales se pueden pasar a la función de trazado a través de el argumento.

Si xyLimits tiene su valor predeterminado NULL, los límites comunes de los ejes x e y están determinados por los valores mínimo y máximo de los argumentos requeridos observado y predicho. Si el argumento opcional refLine es FALSE, la línea de referencia de igualdad discontinua no se agrega al gráfico.

```
#PredictedVsObservedPlot
```

La funcion BasicSummary

Esta función devuelve un marco de datos con una fila para cada columna de df y las siguientes columnas:

1. variable, el nombre de la columna correspondiente de df.
2. tipo, la clase de la variable, según lo definido por la función de clase.
3. niveles, el número de valores distintos exhibidos por la variable.
4. topLevel, el valor más frecuente.
5. topCount, la cantidad de veces que ocurre el valor más frecuente.
6. topFrac, la fracción de registros representada por topCount.

7. missFreq, el número de valores faltantes exhibidos por la variable.
8. missFrac, la fracción de registros representada por missFreq.

```
#BasicSumary
```

La funcion FindOutliers

Aunque es bastante simple en comparación con muchas de las funciones integradas en R, este último ejemplo es el más complejo considerado aquí, y requiere otras cuatro funciones externas para realizar cálculos clave. Específicamente, la función de nivel superior.

Se llama a FindOutliers con el argumento requerido xy tres argumentos de umbral opcionales, pasados a las funciones ThreeSigma, Hampel y BoxplotRule, estas funciones devuelven listas de dos componentes con elementos hacia arriba y hacia abajo, que representan límites de detección de valores atípicos superiores e inferiores para los valores de x calculado por un método diferente FindOutliers llama a la función ExtractDetails con x y estos tres vectores de límite de detección para identificar los puntos individuales en x que se consideran valores atípicos por cada método, y los resultados se combinan en los siguientes cuatro marcos de datos:

1. sumFrame tiene una fila para cada método, dando el nombre del método, el número total de observaciones de datos en x, el número de valores faltantes en x, el número de valores atípicos detectados, los límites de detección de valores atípicos superiores e inferiores, y el mínimo y valores máximos no periféricos.
2. threeFrame proporciona los índices, valores y clase (valores atípicos superiores o inferiores) de cada valor atípico detectado en x según la regla de edición de tres sigmas.
3. HampelFrame proporciona los resultados correspondientes para el identificador de Hampel.
4. boxFrame proporciona los resultados correspondientes para la regla de outlier boxplot.