|  |
| --- |
| Analytic Ideas Services |
| Curso de Programación Avanzada en R. “Aprender a aprender, comprender enseñando, aprender haciendo, trabajar aprendiendo y disfrutar del proceso.” |
| Interno |
|  |

 

**RADV01**

Tabla de contenido

Cómo configurar R 1

Cómo estructurar su código en R 2

Cómo utilizar las funciones internas en R 3

Cómo Obtener valores de Vectores en R 4

Cómo es la indexación de R 4

Cómo extraer valores de un vector en R 5

Funciones 6

Creando funciones 7

Funciones anónimas 8

Closures 9

Tipos de funciones especiales 9

operadores infix 9

Funciones de reemplazo 10

Lexical scoping 10

Return values 12

Function arguments ("formals") 13

Evaluación lenta 13

Ejercicios 15

Entornos 15

introducción 15

Entornos de funciones 16

El entorno en el que fue definina la función. 16

El entorno que se crea cada vez que se ejecuta una función 17

El entorno donde la función vive 18

Modificando y asignando valores 18

El operador <<- 19

Asignación con retraso 19

Enlaces activos 20

Scoping explicito con local. 20

Funciones de primera clase 21

Closures 21

Estado Mutable 22

Funciones de orden superior 23

Manipulación de estructura de datos. 23

Funciones matemáticas de orden superior 25

Listas de funciones 26

Caso de estudio: Integración numérica 30

Ejercicios 33

Sistemas orientados a objetos (OO): S3, S4 y R5 33

S3 33

Clase de objeto 34

Funciones genéricas y método de envío 34

Metodos 35

Genéricos internas 36

Herencia 36

Buenas prácticas 38

S4 38

Clases e instancias 38

Funciones genéricas y métodos 41

En su hábitat natural 46

R5 48

Clases e instancias 48

Métodos 50

En paquetes 51

En su habitat natural 51

SoftwareSystems 52

Cómo seguir con éxito convenciones de nomenclatura en R 56

Cómo elegir un nombre correcto en R 56

Elegir nombres en R que añaden significado a su código 56

Cómo depurar el código en R 57

Conozca lo que debe buscar en R 57

Cómo decirle a R la función para Debug 58

Cómo paso un Debugging sobre funciones en R 59

Cómo cazar Bugs en R 60

Calcular el logit 60

Saber de dónde proviene un error 61

Cómo saber cuándo cuidar de las advertencias en R 61

Cómo leer errores y advertencias en R 63

Cómo generar sus propios mensajes de error en R 65

Cómo crear mensajes de error en R 65

Cómo crear mensajes de advertencia en R 66

El problema 66

Elegir basado en un vector lógico en R 67

Cómo funciona 67

Prúebelo 67

Cómo encadenar sentencias If…Else en R 68

Cómo utilizar bucles con índices en R 69

Cómo recorrer valores en R 70

Construir un bucle for 70

Calcular los valores en un bucle for 71

Usar los valores del vector 71

Cómo hacer más con Bucles en R 72

Cómo vectorizar sus funciones en R 72

Cómo usar la familia Apply de funciones en R 73

Las características de la familia 74

Conozca a tres de los miembros 74

Cómo aplicar funciones a un vector en R 75

Cómo aplicar funciones en filas y columnas en R 76

Contar en R usando la función apply 76

Añadir argumentos extra a la función apply 77

Cómo simplificar los resultados (o no) con la función de sapply en R 78

Cómo trabajar con archivos y carpetas en R 78

Cómo manipular archivos en R 80

Cómo leer datos desde Excel a R 81

Cómo utilizar read.csv() para importar datos en R 82

# Cómo configurar R

Además de aceptar las opciones en el procedimiento de instalación de R, usted puedes cambiar un número de opciones de inicio, adaptando el archivo **Rprofile**.**site**. Este archivo se encuentra dentro del directorio de instalación, en la subcarpeta **…/R-n.n.n/etc**(por ejemplo, **…/R-2.14.1/etc**). El archivo es llamado por R en el arranque, por lo que todo el código R en este archivo se lleva a cabo.

La instalación por defecto de R contiene un archivo **Rprofile.site** válido, por lo que tiene que cambiar esta opción si desea personalizar su inicio.

**Rprofile.site**  es un archivo de texto normal, por lo que se puede editar como lo haría con cualquier otro archivo de texto. El archivo ya contiene algunas opciones que están comentadas, para que tenga una idea de lo que es posible cuando se abre el archivo en un editor de texto (como el Bloc de notas). Asegúrese de consultar la página de ayuda **?options** para obtener más información sobre todas las opciones posibles.

Usted puede personalizar R mucho más mediante la adición de un archivo de código llamado .**Rprofile** a la carpeta de inicio personal. Usted puede encontrar esta carpeta dentro de R estableciendo el directorio de trabajo a "~/", así:

setwd("~/")

getwd()

[1] "C:/Users/.../Documents

La adición de un archivo.**Rprofile** no es necesario, pero R siempre buscará uno, ya sea en la carpeta desde la que llama a R o en el directorio de usuario. Considerando que un archivo **Rprofile**.**site** está vinculado a una instalación específica de R, el archivo.**Rprofile** puede diferir para cada usuario en el mismo sistema.

Si actualiza R, puede dejar el archivo.**Rprofile** donde está y la nueva versión de R automáticamente encontrará y aplicará las opciones allí especificadas. Así que, después de actualizar R a la última versión, hay que adaptar el **Rprofile**.**site** nuevo si desea personalizar R.

Las funciones que usted define o los objetos que se crean con código en **Rprofile.site** no serán visible si usa **ls**(), aunque se puede utilizar sin problemas. Esto también significa que no se pueden eliminar fácilmente del workspace.

Un archivo **Rprofile**.**site**ó .**Rprofile** puede verse como el ejemplo siguiente:

# Sample profile file

# Set CRAN mirror to a default location

options(repos = "http://cran.uk.r-project.org")

# R interactive prompt

options(prompt="R: ")

# sets work directory back to original

go.home <- function() setwd("D:/MyWorkspace")

Con este archivo, R se iniciará con un diferente prompt (**R:** en vez de  **>**) y configurará el servidor del Reino Unido como el servidor por defecto desde el cual se instalarán los paquetes. También se define la función **go.home()**, que se puede utilizar en cualquier momento para establecer el directorio de trabajo de nuevo a su directorio de inicio (**D:/MyWorkspace**, por ejemplo.)

# Cómo estructurar su código en R

Los nombres no son las únicas cosas que pueden influir en la legibilidad del código R. Cuando usted necesite de funciones de anidación o realizar cálculos complejos, el código puede convertirse en un gran lío de texto y símbolos con bastante rapidez.

Por suerte, hay algunos trucos para aclarar su código y ser capaz de descifrar lo que hizo hace meses.

El anidamiento de funciones y cálculos complejos pueden llevar a largas líneas de código. Si usted desea crear un vector con los nombres de sus tres canciones preferidas, por ejemplo, ya está en problemas. Por suerte, R le permite romper una línea de código en varias líneas en su script, por lo que no tiene que desplazarse a la derecha todo el tiempo.

Usted ni siquiera tiene que usar una notación o carácter especial. R sabrá que la línea no está terminada, siempre y cuando usted le da alguna pista. Por lo general, usted tiene que asegurarse de que el comando esta incompleto. Hay varias formas de hacer esto:

* **Puede utilizar un signo de comillas para iniciar una cadena**. R tendrá toda la entrada siguiente incluyendo los saltos de línea como parte de la cadena, hasta que encuentre la marca de coincidencia.
* **Puede finalizar la línea incompleta con un operador (como +, /, <-, y así sucesivamente).** R sabrá que algo más tiene que seguir. Esto le permite crear una estructura en cálculos más largos.
* **Puede abrir un paréntesis para una función.** R leerá toda la entrada hecha en una sola línea hasta que encuentre el paréntesis correspondiente. Esto le permite alinear los argumentos por debajo de una función, por ejemplo.

El siguiente script muestra estas técnicas:

baskets.of.Geraldine <-

c(5,3,2,2,12,9)

baskets.of.Granny <-

c(5,3,2,2,12,9)

Intro <- "It is amazing! The All Star Grannies scored

a total of"

Outro <- "baskets in the last six games!"

Total.baskets <- baskets.of.Granny +

baskets.of.Geraldine

Text <- paste(Intro,

sum(Total.baskets),

Outro)

cat(Text)

Puede copiar este código en un archivo de script y ejecutarlo en la consola. Si ejecuta este pequeño fragmento de código, verá la siguiente salida en la consola:

It is amazing! The All Star Grannies scored

a total of 66 baskets in the last six games!

La función cat(), imprime lo que usted da como argumento directamente a la consola. También interpreta los caracteres especiales como saltos de línea y tabuladores. Si nos fijamos en el vector **Text**, obtenemos:

Text

[1] "It is amazing! The All Star Grannies scored \na total of 71 baskets in the last six games!"

La \n representa el salto de línea. Aunque está pegado a la a, R reconocerá \n como un carácter separado.

Todo esto también trabaja en la línea de comandos. Si usted escribe un comando sin terminar, R va a cambiar el comando a un signo **+**, que indica que puede seguir escribiendo el comando:

cat('If you doubt whether it works,

+ just try it out.')

If you doubt whether it works,

just try it out.

RStudio añade automáticamente un salto de línea al final de una declaración **cat()** si no hay ninguno, pero R no hace eso. Por lo tanto, si usted no utiliza RStudio, recuerde agregar un salto de línea (o el símbolo \n) al final de la cadena.

# Cómo utilizar las funciones internas en R

A veces, puede ser muy útil usar las funciones internas en R. Escribir funciones, de tal manera que necesiten objetos en el entorno global en realidad no tiene sentido, porque el uso de funciones es para evitar la dependencia de objetos en el ambiente global.

De hecho, todo el concepto detrás de R se opone fuertemente al uso de variables globales que se utilizan en diferentes funciones. Como un lenguaje de programación funcional, una de las ideas principales de R es que el resultado de una función no debe ser dependiente de nada sino de los valores para los argumentos de la función. Si le damos a los argumentos los mismos valores, siempre se obtiene el mismo resultado.

Si usted está acostumbrado a otros lenguajes de programación como Java, esta característica puede parecer extraña, pero tiene sus méritos. A veces es necesario repetir algunos cálculos un par de veces dentro de una función, pero estos cálculos sólo tienen sentido dentro de esa función.

Suponga que desea comparar la producción de luz de unas lámparas de potencia media y potencia máxima. Las cortinas que se colocan delante de la ventana para bloquear el sol no están a la misma altura, por lo que se puede medir la cantidad de luz que llega. Usted quiere restar la media de este valor a partir de los resultados con el fin de corregir las mediciones.

Para calcular la eficiencia de energía del 50 por ciento, puede utilizar la siguiente función:

calculate.eff <- function(x, y, control){

min.base <- function(z) z - mean(control)

min.base(x) / min.base(y)

}

Dentro de la función **calculate.eff()** , puede ver otra definición de función para una función **min.base()**. Exactamente como en el caso de otros objetos, esta función se crea en el entorno local de **calculate.eff()**y se destruye cuando la función se lleva a cabo. Usted no encontrará **min.base()** de nuevo en el área de trabajo.

Puede utilizar la función de la siguiente manera:

> half <- c(2.23, 3.23, 1.48)

> full <- c(4.85, 4.95, 4.12)

> nothing <- c(0.14, 0.18, 0.56, 0.23)

> calculate.eff(half, full, nothing)

[1] 0.4270093 0.6318887 0.3129473

Si nos fijamos un poco más de cerca en la definición de la función de min.base(), se observa que ésta utiliza un objeto control pero no tiene un argumento con ese nombre. ¿Cómo funciona esto entonces? Cuando se llama a la función, ocurre lo siguiente:

1. La función **calculate.eff()** crea un nuevo entorno local que contiene los objetos **x** (con el valor de **fifty** y con el valor de **hundred**), **control**  (con el valor de **nothing**), así como la función **min.base()**.
2. La función **min.base()** crea un nuevo entorno local dentro de **calculate.eff()** conteniendo sólo un objeto **z**  con el valor de **x**.
3. **min.base()** busca el objeto control  en el entorno de **calculate.eff()**  y resta la media de este vector a partir de cada número de **z**. Este valor se devuelve.
4. Lo mismo sucede otra vez, pero esta vez **z**  toma el valor de **y**.
5. Ambos resultados se dividen por el otro, y **result**  se transmite de nuevo al ambiente global.

El entorno local está integrado en el entorno en el que la función es definida, no donde llamada. Supongamos que usamos **addPercent()** dentro **calculate.eff()**para dar formato a los números. El entorno local creado por **addPercent()**  no está integrado en el de **calculate.eff()** , sino en el entorno global, donde **addPercent()**  es definido.

# Cómo Obtener valores de Vectores en R

Los vectores serían muy poco prácticos si no se podrian mirar hacia atras y manipular los valores individuales. Estas tareas se pueden realizar con facilidad mediante el uso del sistema de indexación de gran alcance de R.

## 

## Cómo R hace la indexación

Cada vez que R muestra un vector, muestra un número, por ejemplo [1] delante de la salida. En este ejemplo, [1] indica donde está la primera posición en su vector.

Este número se denomina el ***índex*** de ese valor. Si hace un vector más largo — por ejemplo, con los números del 1 al 30 — se ven más índices. Considere este ejemplo:

numbers <- 30:1

numbers

[1] 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14

[18] 13 12 11 10 9 8 7 6 5 4 3 2 1

Aquí, se ve que R cuenta 13 como el valor 18 en el vector. Al comienzo de cada línea, R le indica el índice del primer valor en esa línea.

Si intenta este ejemplo en el equipo, puede aparecer un índice diferente al comienzo de la línea, en función de la anchura de la consola.

## Cómo extraer valores de un vector en R

Los corchetes ([]) ilustran otro punto fuerte de R. Representan una función que puede utilizar para extraer un valor de ese vector. Usted puede obtener el quinto valor del vector de números anterior de esta manera:

numbers[5]

[1] 26

Bien, este ejemplo no es muy impresionante, pero la función bracket (corchetes) toma vectores como argumentos. Si desea seleccionar más de un número, sólo tiene que proporcionar un vector de índices como argumento dentro de los paréntesis, así:

numbers[c(5,11,3)]

[1] 26 20 28

R retorna un vector con los números en el orden que usted pidió. Por lo tanto, se pueden utilizar los índices para ordenar los valores de la forma que desee.

También puede almacenar los índices que desea recuperar en otro vector y dar ese vector como un argumento, como en el siguiente ejemplo:

indices <- c(5,11,3)

numbers[indices]

[1] 26 20 28

También puede utilizar índices para retirar los valores de un vector. Si desea todos los números excepto el tercer valor, usted puede hacerlo con el siguiente código:

numbers[-3]

[1] 30 29 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13

[18] 12 11 10 9 8 7 6 5 4 3 2 1

Aquí, también, se puede utilizar un vector completo de índices. Si quiere retirar los primeros 20 números, utilice este código:

numbers[-(1:20)]

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

Tenga cuidado de agregar paréntesis alrededor de la secuencia. Si no lo hace, R interpretará en el sentido de la secuencia de –1 a 20, que no es lo que se desea. Si ejecuta el código, se obtiene el siguiente mensaje de error:

numbers[-1:20]

Error in numbers[-1:20] : only 0's may be mixed with negative subscripts

Este mensaje hace que se pregunte qué es el índice 0. Bueno, es literalmente nada. Si es el único valor en el vector de índice, se obtiene de regreso un vector vacío, o de longitud cero.

No se puede mezclar valores de índice positivos y negativos, así que seleccionamos una cantidad de valores o bien los retiramos.

Usted puede hacer mucho más con los índices – esto le ayudara a escribir código conciso y rápido.

# Funciones

Si usted está leyendo este documento, usted probablemente ya ha creado muchas funciones en R, y está familiarizado con los conceptos básicos de cómo funcionan. Esto le ayudará a concretar su conocimiento de funciones, y le ayudará a entender algunos de los detalles fundamentales más importantes.

El conocimiento de este capítulo es un pilar importante para hacer cosas útiles con funciones.

Todas las funciones tienen cuatro componentes

* **body(f)**: el objeto citado representa el código dentro de la función
* **formals**(f): la lista de argumentos para la función
* **environment(f)**: el entorno en el cual la función fue definida
* **attr(f, "srcref")**: el código fuente usado para crear la función

(Hay una excepción a esta regla: las funciones primitivas, que utilizan **.Primitive** para llamar código C directamente son manipuladas especialmente: **e.g. formals(sum),body(sum), environment(sum))**

Esto también se puede utilizar para modificar la estructura de la función y su forma de asignación.

Cuando usted imprime una función en R, esta muestra los formals, el código fuente y el entorno. Si el entorno no es desplegado, significa que la función fue creada en el entorno global.

f <- function(x) x

f

# function(x) x

environment(f)

# <environment: R\_GlobalEnv>

print

# function (x, ...)

# UseMethod("print")

# <environment: namespace:base>

Las funciones en R son creados por **function**. Se componen de una lista de argumentos (que puede incluir valores por defecto), y un cuerpo de código que debe ejecutarse al ser evaluado. En **R** los argumentos son pasados por valor, por lo que la única manera que una función puede afectar el exterior es a través de su valor de retorno:

f <- function(x) {

x$a <- 2

}

x <- list(a = 1)

f()

x$a

Las funciones pueden devolver un solo valor, pero esto no constituye una limitación en la práctica, porque siempre se puede devolver una lista que contenga cualquier número de objetos.

Cuando se llama a una función se puede especificar argumentos por posición o por nombre:

mean(1:10)

mean(x = 1:10)

mean(x = 1:10, trim = 0.05)

mean(trim = 0.05, x = 1:10)

Los argumentos se comparan en primer lugar por el nombre exacto, luego por correspondencia de prefijo y finalmente por posición.

## Creando funciones

La herramienta que usamos para crear funciones es **function**. Está muy cerca de ser una función ordinaria R, pero tiene una sintaxis especial: el último argumento de la función esta fuera de la llamada y proporciona el cuerpo de la nueva función. Si no asignamos el resultado de **function** a una variable obtenemos una función anónima:

function(x) 3

# function(x) 3

## Funciones anónimas

En R, las funciones son objetos en su derecho propio. A diferencia de muchos otros lenguajes de programación, las funciones no estan enlazadas automáticamente a un nombre: ellas pueden existir independientemente. Puede que usted ya haya observado esto, porque al crear una función, se utiliza el operador de asignación habitual para darle un nombre.

Dado el nombre de una función como una cadena, usted puede encontrar esa función utilizando **match.fun**. Lo contrario no es posible: porque no todas las funciones tienen un nombre, o las funciones pueden tener varios nombres. Las funciones que no tienen un nombre se llaman **funciones anónimas**.

Usted puede llamar a funciones anónimas, pero el código tiene su dificultad al leer ya que tiene que usar paréntesis en dos formas distintas: para llamar a una función, y para dejar en claro que queremos llamar a la función anónima **function(x) 3** y no dentro de nuestra función anónima llamar a una función llamada **3** (no es un nombre de función válido):

(function(x) 3)()

# [1] 3

# Exactly the same as

f <- function(x) 3

f()

function(x) 3()

# function(x) 3()

La sintaxis se extiende de una manera sencilla si la función tiene parámetros

(function(x) x)(3)

# [1] 3

(function(x) x)(x = 4)

# [1] 4

Al igual que todas las funciones en R, las funciones anónimas tienen **formals**, **body** y **enviroment**

formals(function(x = 4) g(x) + h(x))

# $x

# [1] 4

body(function(x = 4) g(x) + h(x))

# g(x) + h(x)

environment(function(x = 4) g(x) + h(x))

# <environment: R\_GlobalEnv>

## Closures

"Un objeto es una data con funciones. Una clousures es una función con data". --- John D. Cook,

Las funciones anónimas son muy útiles en combinación con clousures, “una función escrita por otra función”. Los clousures son llamadas así porque ellas **encierran** el entorno de la función padre, y pueden acceder a todas las variables y parámetros en esa función. Esto resulta útil ya que nos permite disponer de dos niveles de parámetros. Un nivel de parámetros (el padre) que controla cómo trabaja la función. El otro nivel (el hijo) hace el trabajo. El siguiente ejemplo muestra cómo se puede usar esta idea para generar una familia de funciones potencia. La función padre (**power**) crea las funciones hijas (**square** y **cube**) que hacen el trabajo duro.

power <- function(exponent) {

function(x) x ^ exponent

}

square <- power(2)

square(2) # -> [1] 4

square(4) # -> [1] 16

cube <- power(3)

cube(2) # -> [1] 8

cube(4) # -> [1] 64

Una propiedad interesante de las funciones en R es que básicamente cada función en R es un clousure, ya que todas las funciones recuerdan el entorno en el cual fueron creadas, por lo general ya sea el entorno global, si se trata de una función que usted a escrito, o un entorno paquete, si se trata de una función que alguien la ha escrito.

Pero esto no sirve de mucho puesto que las funciones en R dependen del lexical scoping:

f(1)

# Error in f(1) : could not find function "+"

## Tipos de funciones especiales

Existen dos tipos de funciones especiales en R: operadores infix, y funciones de reemplazo.

### operadores infix

La mayoría de funciones son operadores "prefix": el nombre de la función precede a los argumentos. En R también se pueden crear funciones infix donde el nombre de la función se encuentran entre sus argumentos. Todas los nombres de las funciones infix deben comenzar y terminar con **%**.

Por ejemplo, se podría crear un nuevo operador para combinar cadenas:

"%+%" <- function(a, b) paste(a, b)

"new" %+% "string"

Observe que usted tiene que poner el nombre de la función entre comillas ya que es un nombre especial.

### Funciones de reemplazo

Las funciones de reemplazo actúan como si ellas modificaran sus argumentos en el lugar, y tienen el nombre especial **xxx <-**. Por lo general tienen dos argumentos (**x** y **value**), aunque pueden tener más, y ellos deben retornar el objeto modificado.

Por ejemplo, la siguiente función te permite modificar el segundo elemento de un vector:

"second<-" <- function(x, value) {

x[2] <- value

x

}

x <- 1:10

.Internal(inspect(x))

second(x) <- 5L

x

.Internal(inspect(x))

A menudo resulta útil combinar reemplazar y generar subconjuntos:

x <- setNames(1:3, letters[1:3])

names(x)[2] <- "two"

names(x)

Esto trabaja porque **names(x) [2] <- "two"** se evalúa como **x <- "names<-" (x, "[<-" (names (x), 2, "two"))**, es decir, es equivalente a:

y <- names(x)

y[2] <- "two"

names(x) <- y

Por lo general, la modificación en lugar no creará una copia de los datos, pero si usted está dependiendo de esto para un alto rendimiento, es mejor verificar dos veces.

## Lexical scoping

Scoping es el conjunto de reglas que gobiernan cómo R busca el valor de un símbolo o nombre. Esto es, scoping es el conjunto de reglas que R aplica para ir desde el símbolo **x**, a su valor **10** en el ejemplo siguiente.

x <- 10

x

# [1] 10

R tiene dos tipos de scoping: **lexical scoping**, implementado automáticamente a nivel de lenguaje y el **dynamic scoping**, que se utiliza en funciones seleccionadas para ahorrar tipeo durante el análisis interactivo. Este documento describe lexical scoping, así como los entornos (la estructura de datos que esta detrás de). El dynamic scoping es descrito en el contexto del controlando la evaluación.

Comprender scoping le permite:

* desarrollar herramientas por composición de funciones, según lo descrito en funciones de primera clase
* anular las habituales reglas de evaluación y cálculo en el lenguaje

Lexical scoping busca hasta valores de símbolos basado en como las funciones son anidadas cuando ellas fueron escritas, no cuando fueron llamados. Con ámbito léxico, se puede determinar si el valor de cada variable se buscará sólo observando la definición de la función, usted no necesita saber nada sobre cómo la función es llamada.

El "lexical" en lexical scoping no corresponde a la definición usual ("de o relacionados con palabras o el vocabulario de una lengua que se distingue de su gramática y construcción"), proviene de la ciencias de la computación del término "Lexing", que es parte del proceso que convierte el código representado como texto a piezas significativas que el lenguaje de programación entiende. Es lexical en este sentido, ya que sólo necesitamos la definición de las funciones, y no como ellas son llamadas.

El siguiente ejemplo ilustra el principio básico:

x <- 5

f <- function() {

y <- 10

c(x = x, y = y)

}

f()

# x y

# 5 10

Lexical scoping es la regla que determina que valores son buscados, y no cuándo. A diferencia de otros lenguajes, R mira a los valores en tiempo de ejecución, y no cuando la función se creo. Esto significa que resultados de una función puede ser diferentes dependiendo de los objetos fuera de su entorno:

x <- 15

f()

# x y

# 15 10

x <- 20

f()

# x y

# 20 10

Generalmente, este comportamiento debe ser evitado. Para detectar esta situación, se puede utilizar **codetools :: findGlobals**, que es ejecutada automáticamente por **R CMD check**. Neutralizar manualmente el entorno al entorno vacío no funciona, porque R se basa en el **lexical scoping** para encontrar cada cosa, incluso el operador **+**.

f <- function(x) x + 1

environment(f) <- emptyenv()

f

# function(x)

x + 1

# <environment: R\_EmptyEnv>

Si un nombre es definido dentro de una función, esta ocultará la definición al nivel superior:

g <- function() {

x <- 21

y <- 11

c(x = x, y = y)

}

f()

# x y

# 20 10

g()

# x y

# 21 11

Las mismas reglas se aplican a closures, “funciones que devuelven funciones”:

x <- 10

f <- function() {

y <- 1

function() {

c(x = x, y = y)

}

}

g <- f()

g()

Esto parece un poco mágico (cómo sabe R cuál es el valor de **y** después de que la función ha sido llamada), pero esto se debe a que cada función almacena el entorno en el cual está definido. **Entornos** ofrecen algunos indicadores de cómo usted puede sumergirse y descubrir cuáles son algunos de los valores.

## Return values

Invisable value.

## Function arguments ("formals")

R's function call semantics

Default values.

### Evaluación lenta

Por defecto, los argumentos de la función R son lentos (perezosos) – “ellos no son evaluados cuando usted llama a la función, sino unicamente cuando ese argumento es utilizado”:

f <- function(x) {

10

}

system.time(f(Sys.sleep(10)))

# user system elapsed

# 0 0 0

Si desea garantizar que un argumento es evaluado se puede utilizar **force**:

f <- function(x) {

force(x)

10

}

system.time(f(Sys.sleep(10)))

# user system elapsed

# 0 0 10.001

Los argumentos predeterminados son evaluados en el entorno donde ellos estan definidos. Esto significa que si la expresión depende del entorno actual, los resultados deberían ser diferentes dependiendo de si se utiliza el valor implícito o el proporcionado explícitamente.

f <- function(x = ls()) {

a <- 1

g(x)

}

g <- function(x) {

b <- 2

x

}

f()

f(ls())

Más técnicamente, un argumento sin evaluar es llamado un **promise**, o un thunk. Un promise se ​​compone de dos partes:

* una expresión que da el cálculo diferido, el cual puede ser accesado con **substitute** (ver **control de evaluación** para más detalles)
* el ambiente donde la expresión fue creada y donde esta deberiá ser evaluada

Puede notar que esto es bastante similar a un closure sin argumentos, y en muchos lenguages que no tienen laziness construido como en R, es así como usted podria implementar la laziness.

Esto es particularmente útil en sentencias if:

if (!is.null(x) && y > 0)

Y se puede utilizar para escribir funciones que no serían posibles de otra manera

and <- function(x, y) {

if (!x) FALSE else y }

a <- 1

and(!is.null(a), a > 0)

a <- NULL

and(!is.null(a), a > 0)

Esta función podría no trabajar sin la evaluación lazy porque tanto **x** como **y** deberían siempre ser evaluados, probando si **a> 0**, incluso si **a** era NULL.

Existe un argumento especial llamado **“...”** . Este argumento igualara cualquier argumento que de otra manera no coincidiria, y puede ser utilizado para llamar a otras funciones. Esto es útil si desea recopilar argumentos para llamar a otra función, pero usted no quiere pre-especificar sus posibles nombres.

Para capturar **...** de una forma que sea más fácil trabajar, puede utilizar **list(...)**.

Usar **...** supone un costo – “cualquier argumento mal escrito será ignorado silenciosamente”. Es a menudo mejor ser explícito en lugar de explícito, asi **usted** a su ves podría pedir a los usuarios que proporcionarán una lista de argumentos adicionales. Y esto es ciertamente más fácil si usted está tratando de usar **...** con múltiples funciones adicionales.

## Ejercicios

(Desde el R inferno 8. 2.36): Si **weirdFun()()()** es un comando válido, ¿qué retorna **weirdFun()**?

Escriba un ejemplo.

# Entornos

## introducción

Un **entorno** es muy similar a una lista, con dos diferencias importantes. En primer lugar, un entorno tiene una referencia semántica: La copia habitual de R sobre modificar reglas no se aplican. En segundo lugar, un entorno tiene un padre: si un objeto no es encuentrado en un entorno, entonces R buscará en su padre. Técnicamente, un entorno se compone de un marco, una colección de objetos con nombre (como una lista), y un enlace a un entorno padre.

Usted puede crear entornos con **new.env ()**, ver sus contenidos con **ls ()**, e inspeccionar sus padres con **parent.env ()**:

e <- new.env()

parent.env(e)

ls(e)

e$a <- 1

ls(e)

e$a

Usted puede extraer su contenido con **$** ó **“** o **‘get’.** **‘$’** y **‘[[‘** buscarán únicamente en ese entorno pero **‘get’** buscará también en todos los padres.

b <- 2

e$b

e[["b"]]

get("b", e)

Entornos pueden ser muy útiles en estructuras de datos porque a diferencia de casi cualquier otro tipo de objeto en R, la modificación se lleva a cabo sin una copia. Esto no es algo que deba tomarse a la ligera: se violan la confianza del usuario sobre cómo trabaja el código R, y algunas veces puede ser crítico para el alto rendimiento del código. El siguiente ejemplo muestra cómo se puede usar un entorno para hacer esto. Esto es importante para hacer el entorno padre el entorno vacío y asi que accidentalmente no herede el comportamiento del entorno global.

e <- new.env(parent = emptyenv())

f <- e

exists("a", e)

e$a

ls(e)

# Environments are reference object: R's usual copy-on-modify semantics

# do not apply

e$a <- 10

ls(e)

ls(f)

f$a

Entornos puede ser utilizado como hashes. Mire el paquete **hash** en **CRAN** para un ejemplo. Sin embargo, desde que se añadió [[R5, usted esta generalmente mejor usando clases de referencia en lugar de entornos puros.

Existen pocos entornos especiales que se pueden acceder directamente:

* **globalenv ()**: el área de trabajo del usuario
* **baseenv ()**: el entorno del paquete base
* **emptyenv ()**: el antepasado de todos los entornos

El único entorno que no tiene un padre es **emptyenv ()**, que es el padre eventual de todos los entornos. El entorno más común es el entorno global (**globalenv ()**) que corresponde a la cima para el nivel de trabajo. El padre del entorno global es uno de los paquetes que usted ha cargado (el orden exacto dependerá del orden en el cual los paquetes se han cargado). el padre eventual será el entorno base, que es el entorno de la funcionalidad "base R", que tiene el entorno vacío como padre.

## Entornos de funciones

Los entornos son la estructura de datos que los potestan el alcance. Hay varios entornos asociados a cada función, y es fácil confundirse entre ellos.

* el entorno en el que fue definida la función
* el entorno que se crea cada vez que se ejecuta una función
* el entorno en el que la función vive

### El entorno en el que fue definina la función.

Cuando una función se crea, se obtiene un puntero al entorno donde fue creado. Se puede acceder a este entorno con la función **enviroment**.

x <- 1

f <- function(y) x + y

environment(f)

environment(plot)

environment(t.test)

También es posible modificar el entorno de una función, usando la forma de asignación de **enviroment**. Esto rara vez es útil, pero podemos usarlo para ilustrar en que consiste lo fundamental en la determinacion del alcanze R.

Una de las quejas que la gente a veces realizar sobre R es que la función **f** definida anteriormente debería generar un error, porque no hay ninguna variable **“y”** definida dentro de R. Bueno, se podría solucionar modificando manualmente el entorno de **f** de modo que no pueda encontrar **“y”** dentro del entorno global:

f <- function(x) x + y

environment(f) <- emptyenv()

f(1)

Sin embargo, cuando se ejecuta, no conseguimos el error que esperamos. Debido a que R utiliza sus reglas de alcance consistentemente para todo (incluyendo el mirar funciones), se obtiene un error que **f** no puede encontrar la función **+**.

### El entorno que se crea cada vez que se ejecuta una función

Qué opina que la siguiente función retornara la primera vez que se ejecute? Qué pasara con la segunda vez?

f <- function(x) {

if (!exists("a")) {

message("Defining a")

a <- 1

} else {

a <- a + 1

}

a

}

f()

Puede que se sorprenda que devuelva el mismo valor en todo momento. Esto se debe a que cada vez que una función es llamada, un nuevo entorno está creado para albergar la ejecución. Esto se puede ver con mayor facilidad mediante la devolución del entorno dentro de la función: usando **enviroment()** sin argumentos retorna el actual entorno (intente ejecutarlo en el nivel superior). Cada vez que se ejecuta la función una nueva función es creada. Pero todos ellos tienen el mismo entorno padre - que es el entorno donde la función fue definida.

f <- function(x) {

list(

e = environment(),

p = parent.env(environment())

)

}

f()

f()

### El entorno donde la función vive

El entorno de una función, y el medio ambiente donde vive pueden ser diferentes. En el ejemplo anterior, hemos cambiado el entorno de **f** a que sea del **emptyenv ()**, pero este aún vivía en el **globalenv ().**

El entorno donde la función vive determina cómo hallar la función, el entorno de la función determina cómo se encuentran los valores dentro de la función. Esta importante diferencia es la que permite el paquete **namespaces** para trabajar.

## Modificando y asignando valores

Usted ya conoce los procedimientos estándar para modificar y acceder a los valores en el entorno actual (por ejemplo, **x <- 1; x**). Para modificar valores en otros ambientes tenemos algunas pocas técnicas:

* el tratamiento de entornos como listas

e <- new.env()

e$a <- 1

e$a

* asignar y obtener

e <- new.env()

assign("a", 1, envir = e)

get("a", envir = e)

* evaluar expresiones dentro de un entorno

e <- new.env()

eval(quote(a <- 1), e)

eval(quote(a), e)

# OR

evalq(a <- 1, e)

evalq(a, e)

Por lo general prefiero utilizar la primera forma, porque es muy compacto. Sin embargo, veremos las tres formas en código R en la naturaleza.

### El operador <<-

Otra forma de cambiar los valores es **<<-**. La flecha de asignación regular, **<-**, siempre crea una variable en el entorno actual. La flecha de asignación especial, **<<-**, intente modificar una variable existente al caminar por los entornos padres. Si este no encuentra ninguno, creará una nueva variable en el entorno global.

f <- function() {

g <- function() {

x <<- 2

}

x <- 1

g()

x

}

f()

Si no encuentra una variable existente con ese nombre, se creará una en el medio entorno global. Esto es normalmente indeseable, debido a que las variables globales son por lo general indeseables. Sólo hay un entorno global, por lo que sólo se puede almacenar un solo valor, y este introduce dependencias no evidentes entre funciones.

Volveremos a esta idea en profundidad en **funciones. de primera clase.**

### Asignación con retraso

La asignación tardía es particularmente útil para realizar operaciones costosas que usted no está seguro que va a necesitar. Esta es la esencia de la lazyness - postergar de hacer cualquier trabajo hasta el último minuto posible.

Para crear una variable **x**, que es la suma de los valores de a y b, pero no se evalúada hasta que lo necesitamos, utilizamos **delayedAssign**:

a <- 1

b <- 2

delayedAssign("x", a + b)

a <- 10

x

# [1] 12

**delayedAssign** también proporciona dos parámetros que controlan cuando la evaluación debe ocurrir (**eval.env**) y en cual entorno la variable es asignada en **(assign.env)**.

Autoload es un ejemplo de esto, es un contenedor alrededor de delayedAssign para funciones o datos en un paquete - que hace que R comportarse como si el paquete se carga, pero en realidad no lo carga (es decir, hacer cualquier trabajo) hasta que se llama a una de las funciones. Esta es la forma en que los conjuntos de datos en la mayoría de paquetes trabajo - usted puede llamar a (por ejemplo) los diamantes después de biblioteca (ggplot2) y simplemente funciona, pero no es cargado en memoria a menos que realmente se utilicen.

### Enlaces activos

**makeActiveBinding** le permitirá crear nombres que lucen como variables pero actúan como funciones. Cada vez que accede al objeto se ejecuta una función. Esto le permite hacer cosas como:

makeActiveBinding("x", function(...) rnorm(1), globalenv())

x

# [1] 0.4754442

x

# [1] -1.659971

x

# [1] -1.040291

### Scoping explicito con local.

A veces es útil ser capaz de crear un nuevo alcance sin incrustar en una función. La función **local** le permite hacer exactamente eso - puede ser útil si usted necesita algunas variables temporales para una operación más fácil de entender, pero que luego la quieren eliminar:

df <- local({

x <- 1:10

y <- runif(10)

data.frame(x = x, y = y)

})

**local** ha limitado relativamente su uso (típicamente porque la mayoría del time scoping es mejor utilizando las reglas basadas en funciónes regulares en R) en combinación con **<<-**. Usted puede usar esto si desea crear una variable privada que se comparta entre dos funciones:

get <- NULL

set <- NULL

local({

a <- 1

get <<- function() a

set <<- function(value) a <<- value

})

get()

set(10)

a

get()

Si ha leído **computación-en-el-lenguaje**, debería ser capaz de entender el código fuente del **local**:

local <- function (expr, envir = new.env()) {

eval.parent(substitute(eval(quote(expr), envir)))

}

eval.parent <- function (expr, n = 1) {

p <- parent.frame(n + 1)

eval(expr, p)

}

# Funciones de primera clase

R soporta funciones "de primera clase", funciones que pueden ser:

* creadas anónimamente,
* asignadas a variables y almacenadas en estructuras de datos
* retornadas de funciones (closures),
* pasadas como argumentos a otras funciones (funciones de orden superior)

Ya ha aprendido acerca de funciones anónimas en la sección **funciones**. Este capítulo explorará las otras tres propiedades, y mostrara cómo se puede eliminar la redundancia en el código. El capítulo finaliza con una exploración de la integración numérica, mostrando cómo todas las propiedades de las funciones de primera clase se pueden utilizar para resolver un problema real.

Usted debe estar familiarizado con las propiedades básicas de **scoping y entornos** antes de leer este capítulo.

## Closures

Existen dos funciones incorporadas útiles que retornan closures:

* **Negate** toma una función que retorna un vector lógico, y retorna la negación de esa función. Esto puede ser un útil cuando la función tiene que retorna el opuesto de lo que necesita.

Negate <- function(f) {

f <- match.fun(f)

function(...) !f(...)

}

(Negate(is.null))(NULL)

Esto es muy útil en combinación con funciones de orden superior, como veremos en la siguiente sección.

* **Vectorize** toma una función no vectorizada y vectorisa con respecto a los argumentos dados en el parámetro **vectorise.args**. Esto no le otorga ninguna mejora mágica de rendimiento pero resulta útil si usted quiere una manera rápida de hacer una función vectorizada.

Una extensión moderadamente útil de muestreo sería la de vectorizar con respecto al tamaño: esto permite generar múltiples muestras en una sola llamada.

sample2 <- Vectorize(sample, "size", SIMPLIFY = FALSE)

sample2(1:10, rep(5, 4))

sample2(1:10, 2:5)

En este ejemplo se ha utilizado **SIMPLIFY = FALSE** para garantizar que nuestra nueva función vectorizada siempre retorne una lista. Esto suele ser una buena idea.

**Vectorize** no funciona con funciones primitivas.

* **ecdf**

## Estado Mutable

La capacidad para gestionar las variables en dos niveles permite mantener el estado entre invocaciones a funciones al permitir una función el modificar las variables en el entorno de su padre. La clave para manejar las variables a diferentes niveles es el operador de asignación doble flecha (**<<-**). A diferencia de la asignación usual flecha única **(<-)** que asigna siempre en el entorno actual, el operador doble flecha continuará buscando en la cadena de los entornos de los padres hasta que encuentra un nombre coincidente.

Esto hace posible mantener un contador que registra cuantas veces una función ha sido llamada, como se muestra en el siguiente ejemplo. Cada vez que se ejecuta **new\_counter**,es creado un entorno, inicializa el contador **i** en este entorno, y entonces crea una nueva función.

new\_counter <- function() {

i <- 0

function() {

# do something useful, then ...

i <<- i + 1

i

}

}

La nueva función es una closure, y este entorno es el entorno envolvente. Cuando la closures **counter\_one** y **counter\_two** se ejecutan, cada uno modifica el contador en su entorno envolvente y entonces retorna el actual contador.

counter\_one <- new\_counter()

counter\_two <- new\_counter()

counter\_one() # -> [1] 1

counter\_one() # -> [1] 2

counter\_two() # -> [1] 1

Esta es una técnica importante porque es una forma de generar "estado mutable" en R. R5 amplía esta idea en muchos más detalles.

## Funciones de orden superior

El poder de closures está estrechamente unida a otra importante clase de funciones: funciones de orden superior (HOFs), funciones que toman funciones como argumentos. Los matemáticos distinguen entre los funcionales, que aceptan una función y regresar un escalar, y los operadores de funciónes, que acepta una función y regresar una la función. La integración sobre un intervalo es un funcional, la integral indefinida es un operador de funciónes.

Funciones de orden superior al uso de los programadores R caen en dos campos principales: la manipulación de estructuras de datos y técnicas matemáticas, como se describe a continuación.

### Manipulación de estructura de datos.

La primera familia importante de funciones de orden superior manipulan vectores. Cada una de ellas toma una función como primer argumento, y un vector como su segundo argumento.

Las primeras tres funciones toman un predicado lógico, una función que retorna **TRUE** o **FALSO**. La función de predicado no necesita ser vectorizada, como todas las tres funciones llaman elemento por elemento.

* **Filter**: retorna un nuevo vector que contiene únicamente elementos donde el predicado es TRUE.
* **Buscar**: retorna el primer elemento que coincide con el predicado (o el último elemento si **right = TRUE**).
* **Position**: devuelve la posición del primer elemento que coincide con el predicado (o el último elemento si la right = TRUE).

El siguiente ejemplo muestra algunos usos sencillos:

x <- 200:250

is.even <- function(x) x %% 2 == 0

is.odd <- Negate(is.even)

is.prime <- function(x) gmp::isprime(x) > 1

Filter(is.prime, x)

# [1] 211 223 227 229 233 239 241

Find(is.even, x)

# 200

Find(is.odd, x)

# 201

Position(is.prime, x, right = T)

# 42

Las siguientes dos funciones trabajan con clases más generales de funciones:

* **Map**: puede tomar más de un vector como entrada y **f** llamaelemento por elemento en cada entrada. Retorna una lista.
* **Reduce**: reduce recursivamente un vector a un único valor, primero f llama los primeros dos elementos, entonces el resultado de **f** y el segundo elemento y así sucesivamente.

**if x = 1:5** entonces el resultado sería **f (f (f (f (1, 2), 3), 4), 5)**.

**If right = TRUE**, entonces la función es llamada en el orden inverso: **f (1, f (2, f (3, f (4, 5))))**.

Usted también puede especificar un valor **init** en cuyo caso el resultado sería **f (f (f (f (f (init, 1), 2), 3), 4), 5)**

Reduce es útil para implementar muchos tipos de operaciones recursivas: fusiones, uniones, intersecciones, la búsqueda de los valores más pequeños.

A diferencia de **Map**, la implementación de estos cinco procesamiento de vectores HOFs es directo y animo a leer el código fuente para entender la forma en que cada uno trabaja.

Otras familias de funciones de orden superior incluyen:

* El familia **apply**: **eapply, lapply, mapply, tapply, sapply, vapply, by**. Cada una de estas funciones descompone una estructura de datos de alguna manera, aplica la función a cada pieza y luego los junta de nuevo.
* Las funciones **\*\*ply** del paquete **plyr** que procuran unir la base de funciones apply separando limpiamente en función del tipo de entrada ellos separan y el tipo de salida que producen.
* Las funciones para manipulación de array modifican arrays para calcular distintos márgenes u otros resúmenes, o generalizar la multiplicación de matrices de varias maneras: **apply, outer, kronecker, sweep, addmargins.**

### Funciones matemáticas de orden superior

Funciones de orden superior surgen a menudo en matemáticas. En esta sección exploraremos algunas de las funciones matemáticas integradas HOF en R. Existen tres funciones que trabajan con una función numérica 1d:

* **integrate**: ella integra sobre un determinado intervalo
* **uniroot**: encuentra donde vale a cero sobre un intervalo dado
* **optimise**: encuentrar la ubicación de mínimos (o máximos)

Exploremos cómo éstos se usan con una función simple:

integrate(sin, 0, pi)

uniroot(sin, pi \* c(1 / 2, 3 / 2))

optimise(sin, c(0, 2 \* pi))

optimise(sin, c(0, pi), maximum = TRUE)

Hay una función que trabaja con funciónes numérica n-dimensional, **Optim**, la cual encuentra la ubicación de un mínimo.

En estadística, la optimización es a menudo para la estimación de máxima verosimilitud. La estimación de máxima verosimilitud es una coincidencia natural para las closures ya que los argumentos para una probabilidad caen en dos grupos: los datos, lo cuales se fijan para un problema determinado, y los parámetros, los cuales variarán a medida que tratan de encontrar un máximo numéricamente. Esto naturalmente da lugar a un enfoque como el siguiente:

# Negative log-likelihood for Poisson distribution

poisson\_nll <- function(x) {

n <- length(x)

function(lambda) {

n \* lambda - sum(x) \* log(lambda) # + terms not involving lambda

}

}

nll1 <- poisson\_nll(c(41, 30, 31, 38, 29, 24, 30, 29, 31, 38))

nll2 <- poisson\_nll(c(6, 4, 7, 3, 3, 7, 5, 2, 2, 7, 5, 4, 12, 6, 9))

optimise(nll1, c(0, 100))

optimise(nll2, c(0, 100))

## Listas de funciones

En R, las funciones pueden ser almacenados en listas. Junto con closures y funciones de orden superior, esto nos da un conjunto de potentes herramientas para reducir duplicación en código.

Vamos a empezar con un ejemplo sencillo: benchmarking, cuando se compara el rendimiento de múltiples metodos para el mismo problema. Por ejemplo, si se desea comparar algunos metodos para calcular la media, se puede almacenar cada método (función) en una lista:

compute\_mean <- list(

base = function(x) mean(x),

sum = function(x) sum(x) / length(x),

manual = function(x) {

total <- 0

n <- length(x)

for (i in seq\_along(x)) {

total <- total + x[i] / n

}

total

}

)

Llamar a una función de una lista es sencillo: basta con sacarlo de la lista:

x <- runif(1e5)

system.time(compute\_mean$base(x))

system.time(compute\_mean$manual(x))

Si queremos llamar a todas las funciones para comprobar que lo hemos implementado correctamente y regresar el mismo resultado, podemos utilizar **lapply**, ya sea con una función anónima, o una nueva función que llama este primer argumento con todos los demás argumentos:

lapply(compute\_mean, function(f) f(x))

call\_fun <- function(f, ...) f(...)

lapply(compute\_mean, call\_fun, x)

Podemos llevar el tiempo de todas las funciones en la lista con **lapply** o **Map** junto con una función anónima simple:

lapply(compute\_mean, function(f) system.time(f(x)))

Map(function(f) system.time(f(x)), compute\_mean)

Si funciones de temporización es algo que queremos hacer, podemos añadir otro nivel de abstracción: un closure que automáticamente lleve  el tiempo que tarda una función. Entonces creamos una lista de funciones temporizadas y llamamos a los temporizadores con nuestros **x** especificado.

timer <- function(f) {

force(f)

function(...) system.time(f(...))

}

timers <- lapply(compute\_mean, timer)

lapply(timers, call\_fun, x)

Otro buen ejemplo es cuando queremos resumir un objeto de múltiples maneras. Podemos almacenar cada función de resumen en una lista, y ejecutar cada función con **lapply** y **call\_fun**:

funs <- list(

sum = sum,

mean = mean,

median = median

)

lapply(funs, call\_fun, 1:10)

¿Qué pasa si quisiéramos modificar nuestras funciones de resumen para eliminar automáticamente valores perdidos? Una posibilidad sería hacer una lista de funciones anónimas que requieran nuestras funciones de resumen con los argumentos apropiados:

funs2 <- list(

sum = function(x, ...) sum(x, ..., na.rm = TRUE),

mean = function(x, ...) mean(x, ..., na.rm = TRUE),

median = function(x, ...) median(x, ..., na.rm = TRUE)

)

Pero esto lleva a una gran cantidad de duplicación - cada función es casi idéntica aparte de un nombre de función diferente. Podemos escribir una closure para abstraer esta forma:

remove\_missings <- function(f) {

function(...) f(..., na.rm = TRUE)

}

funs2 <- lapply(funs, remove\_missings)

También se puede tomar un enfoque más general. Una función útil es Curry (nombre de un famoso científico de la computación Haskell Curry, no la comida), la cual implementa "la aplicación de  una función parcial". Lo que la función Curry hace es crear una nueva función que se pasa en los argumentos  especificados. Un ejemplo hará esto más claro:

add <- function(x, y) x + y

addOne <- function(x) add(x, 1)

addOne <- Curry(add, y = 1)

Una forma de implementar Curry es la siguiente:

Curry <- function(FUN,...) {

.orig <- list(...)

function(...) {

do.call(FUN, c(.orig, list(...)))

}

}

(Usted debe ser capaz de averiguar cómo funciona esto. Mire los ejercicios.)

Sin embargo, implementarlo tal como ésta impide que los argumentos sean evaluados lazily, así este tiene una implementación algo más complicada, básicamente trabaja mediante la creación de una función anónima manualmente. Usted debe ser capaz de averiguar cómo funciona esto después de haber leído el capítulo de la **computación en lenguaje**. (Esperemos que esta función seá incluida en una futura versión de R.)

Curry <- function(FUN, ...) {

args <- match.call(expand.dots = FALSE)$...

args$... <- as.name("...")

env <- new.env(parent = parent.frame())

if (is.name(FUN)) {

fname <- FUN

} else if (is.character(FUN)) {

fname <- as.name(FUN)

} else if (is.function(FUN)){

fname <- as.name("FUN")

env$FUN <- FUN

} else {

stop("FUN not function or name of function")

}

curry\_call <- as.call(c(list(fname), args))

f <- eval(call("function", as.pairlist(alist(... = )), curry\_call))

environment(f) <- env

f

}

Pero volvamos a nuestro problema. Con la función de Curry podemos reducir un poco su código:

funs2 <- list(

sum = Curry(sum, na.rm = TRUE),

mean = Curry(mean, na.rm = TRUE),

median = Curry(median, na.rm = TRUE)

)

Pero si miramos más de cerca revelará que sólo estamos aplicando la misma función a cada elemento en una lista, y ese es el trabajo de **lapply**. Esto reduce drásticamente la cantidad de código que necesitamos:

funs2 <- lapply(funs, Curry, na.rm = TRUE)

Pensemos en un caso similar, pero sutilmente diferente. Tomemos un vector de números y generemos una lista de funciones que correspondan a medias truncadas con una cantidad de recorte. El código siguiente no funciona porque queremos que el primer argumento de **Curry** para ser fijado a media. Podríamos intentar especificar el nombre del argumento ya que la correspondencia fija prevalece sobre la posición, pero esto no funciona porque el nombre de la función a llamar en **lapply** también es **FUN**. Y no hay manera de especificar que queremos llamar al argumento **trim**.

trims <- seq(0, 0.9, length = 5)

lapply(trims, Curry, "mean")

lapply(trims, Curry, FUN = "mean")

En lugar de eso puede utilizar una función anónima

funs3 <- lapply(trims, function(t) Curry("mean", trim = t))

lapply(funs3, call\_fun, c(1:100, (1:50) \* 100))

Pero esto no funciona, ya que cada función recibe una promesa para evaluar **t**, y esa promesa no es evaluada hasta que todas las funciones se ejecuten. Para hacer que esto funcione es necesario forzar manualmente la evaluación de **t**:

funs3 <- lapply(trims, function(t) {force(t); Curry("mean", trim = t)})

lapply(funs3, call\_fun, c(1:100, (1:50) \* 100))

Una solución más sencilla en este caso es el uso de **Map**, como se describe en el capítulo anterior, que trabaja de manera similar a **lapply** excepto que puede proporcionar múltiples argumentos tanto por nombre y posición. Para este ejemplo, no es un buen trabajo encontrar la manera de nombrar las funciones, pero eso es fácil de solucionar.

funs3 <- Map(Curry, "mean", trim = trims)

names(funs3) <- trims

lapply(funs3, call\_fun, c(1:100, (1:50) \* 100))

Por lo general es mejor utilizar **lapply** porque es más familiar para la mayoría de los programadores R, y es algo más simple por lo tanto es un poco más rápido.

## Caso de estudio: Integración numérica

Para terminar este capítulo, vamos a desarrollar una simple herramienta de integración numérica, y en el trayecto, ilustrar el uso de muchas propiedades de las funciones de primera clases: haremos uso de funciones anónimas, listas de funciones, funciones que hacen clausuras y funciones que toman funciones como entrada. Cada paso es impulsado por un deseo de hacer nuestro enfoque más general y reducir la duplicación.

Empezaremos con dos métodos muy simples: el punto medio y las reglas trapezoidales. Cada uno toma una función que desea integrar, **f**, y un rango para integrar sobre, **a** a **b**. Para este ejemplo trataremos de integrar **sin x** entre **0** y **pi**, ya que tiene una respuesta simple: 2

midpoint <- function(f, a, b) {

(b - a) \* f((a + b) / 2)

}

trapezoid <- function(f, a, b) {

(b - a) / 2 \* (f(a) + f(b))

}

midpoint(sin, 0, pi)

trapezoid(sin, 0, pi)

Ninguna de estas funciones ofrece una aproximación tan buena, por lo que haremos lo que se hace normalmente en cálculo: dividir el intervalo en intervalos más pequeños e integrar cada pieza usando una de las reglas simples. Para ello creamos dos funciones para llevar a cabo la integración compuesta:

midpoint\_composite <- function(f, a, b, n = 10) {

points <- seq(a, b, length = n + 1)

h <- (b - a) / n

area <- 0

for (i in seq\_len(n)) {

area <- area + h \* f((points[i] + points[i + 1]) / 2)

}

area

}

trapezoid\_composite <- function(f, a, b, n = 10) {

points <- seq(a, b, length = n + 1)

h <- (b - a) / n

area <- 0

for (i in seq\_len(n)) {

area <- area + h / 2 \* (f(points[i]) + f(points[i + 1]))

}

area

}

midpoint\_composite(sin, 0, pi, n = 10)

midpoint\_composite(sin, 0, pi, n = 100)

trapezoid\_composite(sin, 0, pi, n = 10)

trapezoid\_composite(sin, 0, pi, n = 100)

mid <- sapply(1:20, function(n) midpoint\_composite(sin, 0, pi, n))

trap <- sapply(1:20, function(n) trapezoid\_composite(sin, 0, pi, n))

matplot(cbind(mid = mid, trap))

Pero note que existe una mucha duplicación entre **midpoint\_composite** y **trapezoid\_composite**: son básicamente los mismos, aparte de la regla interna que permite integrar en un intervalo simple. Vamos a extraer una función general de integracion compuesta:

composite <- function(f, a, b, n = 10, rule) {

points <- seq(a, b, length = n + 1)

area <- 0

for (i in seq\_len(n)) {

area <- area + rule(f, points[i], points[i + 1])

}

area

}

midpoint\_composite(sin, 0, pi, n = 10)

composite(sin, 0, pi, n = 10, rule = midpoint)

composite(sin, 0, pi, n = 10, rule = trapezoid)

Esta función ahora toma dos funciones como argumentos: la función para integrar y la regla de integración para usar en intervalos simples. Ahora podemos agregar reglas incluso mejores para la integración de intervalos pequeños:

simpson <- function(f, a, b) {

(b - a) / 6 \* (f(a) + 4 \* f((a + b) / 2) + f(b))

}

boole <- function(f, a, b) {

pos <- function(i) a + i \* (b - a) / 4

fi <- function(i) f(pos(i))

(b - a) / 90 \*

(7 \* fi(0) + 32 \* fi(1) + 12 \* fi(2) + 32 \* fi(3) + 7 \* fi(4))

}

Comparemos estos diferentes enfoques.

expt1 <- expand.grid(

n = 5:50,

rule = c("midpoint", "trapezoid", "simpson", "boole"),

stringsAsFactors = F)

abs\_sin <- function(x) abs(sin(x))

run\_expt <- function(n, rule) {

composite(abs\_sin, 0, 4 \* pi, n = n, rule = match.fun(rule))

}

library(plyr)

res1 <- mdply(expt1, run\_expt)

library(ggplot2)

qplot(n, V1, data = res1, colour = rule, geom = "line")

Resulta que las reglas del punto medio, trapecio, Simpson y de Boole son todos ejemplos de una familia más general llamado reglas Newton-Cotes. Podemos llevar nuestra integración un paso más allá mediante la extracción del aspecto común para producir una función que pueda generar cualquier regla de  Newton-Cotes :

# http://en.wikipedia.org/wiki/Newton%E2%80%93Cotes\_formulas

newton\_cotes <- function(coef, open = FALSE) {

n <- length(coef) + open

function(f, a, b) {

pos <- function(i) a + i \* (b - a) / n

points <- pos(seq.int(0, length(coef) - 1))

(b - a) / sum(coef) \* sum(f(points) \* coef)

}

}

trapezoid <- newton\_cotes(c(1, 1))

midpoint <- newton\_cotes(1, open = T)

simpson <- newton\_cotes(c(1, 4, 1))

boole <- newton\_cotes(c(7, 32, 12, 32, 7))

milne <- newton\_cotes(c(2, -1, 2), open = TRUE)

# Alternatively, make list then use lapply

lapply(values, newton\_cotes, closed)

lapply(values, newton\_cotes, open, open = TRUE)

lapply(values, do.call, what = "newton\_cotes")

expt1 <- expand.grid(n = 5:50, rule = names(rules), stringsAsFactors = F)

run\_expt <- function(n, rule) {

composite(abs\_sin, 0, 4 \* pi, n = n, rule = rules)

}

Matemáticamente, el siguiente paso para mejorar la integración numérica consiste en pasar de una malla de puntos uniformemente espaciados a una red donde los puntos están más juntos cerca del final del intervalo.

## Ejercicios

1. Lea el código fuente para **Filter, Negate, Find** y **Position**. Escriba un par de sentencias para cada uno describiendo cómo  funcionan.
2. Escribe una función **And** que reciba dos funciones lógicas, retorne un And lógico de todos sus resultados. Extienda la función para trabajar con cualquier número de funciones lógicas. Escriba funciones similares para **Or** y **Not**.
3. Escriba una función compuesta general que compone junto un número arbitrario de funciones. Escríbalo utilizando tanto la recursividad y el bucle.
4. ¿Cómo funciona la primera parte del trabajo **Curry**?

# Sistemas orientados a objetos (OO): S3, S4 y R5

## S3

Fundamental para cualquier sistema orientado a objetos son los conceptos de clase y método. Una clase define un tipo de objeto, describiendo que propiedades posee, cómo se comporta, y cómo se relaciona con otros tipos de objetos. Cada objeto debe ser una instancia de alguna clase. Un método es una función asociada con un tipo particular de objeto.

S3 implementa un estilo de programación orientada a objetos denominado genéricas funciones OO. Esto es diferente a la mayoría de los lenguajes de programación, como Java, C y C #, que implementan transferencia de mensajes OO. En el estilo transferencia de mensajes, los mensajes (métodos) son enviados a los objetos y los objetos determina qué función llamar. Normalmente, este objeto tiene una aparición especial en la llamada al método, por lo general aparecen delante del nombre del método / mensaje: por ejemplo, canvas.drawRect ("azul"). S3 es diferente. Si bien los cálculos se sigue realizando a través de métodos, un tipo especial de función denominada función genérica decide qué método llamar. Los métodos se definen de la misma forma como una función normal, pero son llamados de una manera diferente, como se verá en breve.

El uso principal de la programación OO en R es para los métodos de impresión, summary y plot. Estos métodos permiten tener una función genérica, por ejemplo print (), que muestra el objeto de manera diferente dependiendo de su tipo: la impresión de un modelo lineal es muy diferente a la impresión de un data frame.

### Clase de objeto

La clase de un objeto viene determinada por su atributo **class**, un vector de caracteres de nombres de clase. El siguiente ejemplo muestra cómo crear un objeto de la clase foo:

x <- structure(1, class = "foo")

La clase se almacena como un atributo, pero es mejor que lo modifique con la función **class()**, ya que esta comunica tu intención más claramente:

class(x) <- "foo"

class(x)

[1]

"foo"

Puede utilizar este método para convertir cualquier objeto en un objeto de la clase "foo", si esto tiene sentido o no. Los objetos no se limitan a una sola clase, y puede tener varias clases:

class(x) <- c("A", "B")

class(x) <- LETTERS

Como se discute en la siguiente sección, R busca métodos en el orden en el que aparecen en el vector de clases. Así que en este ejemplo, sería como la clase A hereda de la clase B - si un método no está definido para A, se recurrirá a B. Sin embargo, si se cambiara el orden de las clases, lo contrario sería cierto! Esto es porque S3 no define ninguna relación formal entre las clases, o incluso alguna definición de lo que una clases individuo es. Usted viene de un ambiente estricto como Java, esto parece bastante aterrador (y lo es) pero sí ofrece a los usuarios una gran cantidad libertad o. Aunque es muy difícil dejar que alguien haga algo que no quieres que hagan, los usuarios no se verán frenados porque hay algo que usted no han implementado todavía.

### Funciones genéricas y método de envío

Método de envío se inicia con una función genérica que decide cual método específico para su expedición al. Funciones genéricas todas las tienen la misma forma: una llamada a UseMethod que especifica el nombre genérico y el objeto del envío, el. Esto significa que las funciones genéricas son generalmente muy simple, como a continuación:

mean <- function (x, ...) { UseMethod("mean", x) }

Los métodos son funciones ordinarias que usan una convención de nomenclatura especial: **generic.class:**

mean.numeric <- function(x, ...) sum(x) / length(x)

mean.data.frame <- function(x, ...) sapply(x, mean, ...)

(Estos son algo versiones del código simplificada real).

Como se puede deducir de este ejemplo, UseMethod utiliza la clase de x para saber cuál método llamar. Si x tiene más de una clase, por ejemplo, c ("foo", "bar"), UseMethod buscaría a mean.foo y si no lo encuentra, entonces buscaría mean.bar. Como opción final, UseMethod buscará un método predeterminado, mean.default, y si eso no existe, se producirá un error. El mismo criterio se aplica sin importar el número de clases que tiene un objeto:

x <- structure(1, class = letters)

bar <- function(x) UseMethod("bar", x)

bar.z <- function(x) "z"

bar(x)

[1]

"z"

Una vez UseMethod ha encontrado el método correcto, se invoca de modo especial. En lugar de crear un entorno de evaluación nueva, se utiliza el entorno de la llamada a la función actual (la llamada al genérico), por lo que cualquier asignación o evaluaciones que se hicieron antes de la llamada a UseMethod estarán accesibles al método. Los argumentos que se utilizaron en la llamada a la genérica se pasan al método en el mismo orden en que fueron recibidos.

Dado que los métodos son funciones normales de R, puede llamar directamente. Sin embargo, usted no debe hacer esto porque se pierden los beneficios de tener una función genérica:

bar.x <- function(x) "x"

bar.x(x)

[1] "x"

bar.z(x)

[1] "z"

### Metodos

Para saber en qué clases una función genérica tiene métodos para, puede utilizar la función métodos. Recuerde que en R, que los métodos están asociados con funciones (no los objetos), por lo que se pasa el nombre de la función, en lugar de la clase, como era de esperar:

methods("bar")

# [1] bar.x bar.z

methods("t")

# [1] t.data.frame t.default t.ts\*

# Non-visible functions are asterisc

Non-visible functions son funciones que no han sido exportadas por un paquete, por lo que necesitará utilizar la función getAnywhere para acceder a ellas si usted quiere ver la fuente.

### Genéricos internas

Algunas funciones internas de C también son genéricas, lo que significa que el método de envío no se realiza por funciones R, sino que está realizado por funciones de C especiales. Es importante conocer qué funciones son internamente genéricas, para que pueda escribir métodos para ellos, y así que estás consciente de las diferencias leves en la método. envío No es fácil determinar si una función es internamente genérica, ya que sólo se ve como una llamada típica a C:

length <- function (x) .Primitive("length")

cbind <- function (..., deparse.level = 1) .Internal(cbind(deparse.level, ...))

Así como la **length** y **cbind**, funciones internas genéricas incluyen dim, c, as.character, name y rep.

La lista completa se puede encontrar en la variable global .S3PrimitiveGenerics, y más detalles se dan en? InternalMethods. Genéricas internas tiene un mecanismo de entrada algo diferente a otras funciones genéricas: antes de probar el método por defecto, intentará enviar a modo de objetos, es decir, el **mode**(x). El siguiente ejemplo muestra la diferencia:

x <- structure(as.list(1:10), class = "myclass")

length(x)

# [1] 10

mylength <- function(x) UseMethod("mylength", x)

mylength.list <- function(x) length(x)

mylength(x)

# Error in UseMethod("mylength", x) :

# no applicable method for 'mylength' applied to an object of class

# "myclass"

### Herencia

La función NextMethod proporciona un mecanismo de herencia simple, usando el hecho de que la clase de un objeto S3 es un vector. Este es un comportamiento muy distinto a la mayoría de otros lenguajes, ya que significa que es posible tener diferentes jerarquías de herencia para objetos diferentes:

baz <- function(x) UseMethod("baz", x)

baz.A <- function(x) "A"

baz.B <- function(x) "B"

ab <- structure(1, class = c("A", "B"))

ba <- structure(1, class = c("B", "A"))

baz(ab)

baz(ba)

**NextMethod ()** funciona como **UseMethod** pero en lugar de enviar sobre el primer elemento del vector de clases, su envío será basado en el segundo (o subsiguiente) elemento:

baz.C <- function(x) c("C", NextMethod())

ca <- structure(1, class = c("C", "A"))

cb <- structure(1, class = c("C", "B"))

baz(ca)

baz(cb)

Los detalles exactos son un poco más difícil: NextMethod no trabaja realmente con el atributo de clase del objeto, se utiliza una variable global (.Class) para llevar un registro de cuál clase llamar siguiente. Esto significa que cambiar manualmente la clase del objeto no tendrá ningún impacto sobre la herencia:

# Turn object into class A - doesn't work!

baz.D <- function(x) {

class(x) <- "A"

NextMethod()

}

da <- structure(1, class = c("D", "A"))

db <- structure(1, class = c("D", "B"))

baz(da)

baz(db)

Métodos invocados como resultado de una llamada a **NextMethod** se comportan como si se hubiesen invocado desde el método anterior. Los argumentos para el método heredado se encuentran en el mismo orden y tienen los mismos nombres que la llamada al método actual, y por lo tanto son los mismos que la llamada a la genérica. Sin embargo, las expresiones para los argumentos son los nombres de los argumentos correspondientes formales del método actual. Así, los argumentos tienen valores que corresponden a su valor en el momento en que se invocó **NextMethod**. Argumentos sin evaluar permanecen sin evaluar. argumentos Que faltan siguen desaparecidos.

Si **NextMethod** es llamado en una situación donde no hay segunda clase devolverá un error. Una selección de estos errores se muestra a continuación para saber qué buscar.

c <- structure(1, class = "C")

baz(c)

# Error in UseMethod("baz", x) :

# no applicable method for 'baz' applied to an object of class "C"

baz.c(c)

# Error in NextMethod() : generic function not specified

baz.c(1)

# Error in NextMethod() : object not specified

### Buenas prácticas

Cree un método de construcción que compruebe el tipo de la entrada, y devuelve una lista con la etiqueta de clase correcta.new\_xxx <- función (...) {}

Escriba una función para comprobar si un objeto es de tu clase: is.XXX <- función (x) hereda (x, "XXX")

Cuando se implementa un vector de clases, se deben implementar los siguientes métodos: length, [, [<-

Cuando se implementa una clase matrix / array, se deben implementar estos métodos: dim (recupera nrow y ncol), t, dimnames (recupera rownames y colnames), dimnames <- (recupera colnames<-, rownames <-).

## S4

En comparación con S3, el sistema de objetos S4 es mucho más estricto, y mucho más cercano a otros sistemas OO. Le recomiendo que se familiarice con la forma en que S3 trabaja antes de leer este documento - muchas de las ideas fundamentales son las mismas, pero la aplicación es mucho más estricta. Hay dos diferencias principales de S3:

* definiciones formales de clases: a diferencia de S3, S4 define formalmente la representación y la herencia para cada clase
* múltiple expedición: la función genérica puede ser enviado a un método basándose en la clase de cualquier número de argumentos, no sólo una

Aquí se introducen los fundamentos de S4, tratando permanecer lejos de la esoterica y centrándose en las ideas que usted necesita para comprender y escribir la mayor parte de código S4.

### Clases e instancias

En S3, puede convertir cualquier objeto en un objeto de una determinada clase sólo por establecer el atributo class. S4 es más estricto: hay que definir la representación de la llamada utilizando **setClass**, y la única manera de crear es a través de la función constructor **new**.

Una clase tiene tres propiedades fundamentales:

* un nombre: una cadena alfanumérica que identifica la clase
* representación: una lista de slots (o atributos), entregan sus nombres y clases. Por ejemplo, una clase de persona podría ser representado por un nombre de tipo caracteres y una edad de tipo numérica, de la siguiente manera: la representation (name = "character", edad = "numeric")
* un vector caracter de clases que hereda de, o en terminología S4, **contiene**. Observe que S4 soporta herencia múltiple, pero esto debe ser usado con cuidado, ya que hace métodos de búsqueda extremadamente complicado.

Se crea una clase con **setClass**:

setClass("Person", representation(name = "character", age = "numeric")) setClass("Employee", representation(boss = "Person"), contains = "Person")

y se crea una instancia de una clase con **new:**

hadley <- new("Person", name = "Hadley", age = 31)

A diferencia de S3, S4 comprobaciones que todos los slots tengan el tipo correcto:

hadley <- new("Person", name = "Hadley", age = "thirty")

# invalid class "Person" object: invalid object for slot "age" in class

# "Person": got class "character", should be or extend class "numeric"

hadley <- new("Person", name = "Hadley", sex = "male")

# invalid names for slots of class "Person": sex

Si usted omite un slot, se lo inicia con el objeto por defecto de la clase.

Para tener acceso a slots de un objeto S4 se utiliza @ no, $:

hadley <- new("Person", name = "Hadley")

hadley@age

# numeric(0)

O si usted tiene una cadena de caracteres que da un nombre de slot, se utiliza la función **slot**:

slot(hadley, "age")

Este es el equivalente de **[[.**

Un valor vacío para la edad probablemente no es lo que quieres, así que puedes también asignarle un prototipo por defecto para la clase:

setClass("Person", representation(name = "character", age = "numeric"), prototype(name = NA\_character\_, age = NA\_real\_))

hadley <- new("Person", name = "Hadley")

hadley@age

# [1] NA

**getSlots** devolverá una descripción de todos los slots de la clase:

getSlots("Person")

# name age

# "character" "numeric"

Usted puede averiguar la clase de un objeto con **is**.

Tenga en cuenta que hay una cierta tensión entre el habitual estilo interactivo funcional de R y los globales efectos secundarios que causan las definiciones de clase S4. En la mayoría de lenguajes de programación, definición de clases tiene lugar en tiempo de compilación, mientras que la instanciación de objetos tiene lugar en tiempo de ejecución - es inusual al poder crear nuevas clases de forma interactiva. En particular, cabe destacar que los ejemplos se basan en el hecho de que múltiples llamadas a **setClass** con el mismo nombre de clase silenciosamente anulará la definición anterior, a menos que la primera definición está sellada con **sealed = TRUE**.

#### Comprobación de validez

Usted también puede proporcionar un método opcional que aplica restricciones adicionales. Esta función debe tener un solo argumento llamado object y debería devolver TRUE si el objeto es válido, y si no debería devolver un vector de caracteres dando todas las razones que no es válido.

check\_person <- function(object) {

errors <- character()

length\_age <- length(object@age)

if (length\_age != 1) {

msg <- paste("Age is length ", length\_age, ". Should be 1", sep = "")

errors <- c(errors, msg)

}

length\_name <- length(object@name)

if (length\_name != 1) {

msg <- paste("Name is length ", length\_name, ". Should be 1", sep = "")

errors <- c(errors, msg)

}

if (length(errors) == 0) TRUE else errors

}

setClass("Person", representation(name = "character", age = "numeric"), validity = check\_person)

new("Person", name = "Hadley")

# invalid class "Person" object: Age is length 0. Should be 1

new("Person", name = "Hadley", age = 1:10)

Error in validObject(.Object) :

invalid class "Person" object: Age is length 10. Should be 1

# But note that the check is not automatically applied when we modify

# slots

directly hadley <- new("Person", name = "Hadley", age = 31)

hadley@age <- 1:10

# Can force check with validObject:

validObject(hadley)

# invalid class "Person" object: Age is length 10. Should be 1

### Funciones genéricas y métodos

Funciones genéricas y métodos trabajan de forma similar a S3, pero el envío se basado en el conjunto de todos argumentos, y hay una sintaxis especial para la creación de funciones genéricas y métodos nuevos.

La función **setGeneric** proporciona dos maneras de principales de crear una genérica nueva. Usted puede convertir una función existente a una función genérica, o puede crear una nueva desde cero.

sides <- function(object) 0

setGeneric("sides")

Si usted crea su cuenta, el segundo argumento de **setGeneric** debería ser una función que define todos los argumentos que deseas enviar a y contiene una llamada a **standardGeneric**:

setGeneric("sides", function(object) {

standardGeneric("sides")

})

El ejemplo siguiente establece una jerarquía simple de formas para utilizar con la función **sides**.

setClass("Shape")

setClass("Polygon", representation(sides = "integer"), contains = "Shape")

setClass("Triangle", contains = "Polygon")

setClass("Square", contains = "Polygon")

setClass("Circle", contains = "Shape")

Definir un método para polígonos es sencillo: sólo usamos la ranura **sides**. La función **setMethod** tiene tres argumentos: el nombre de la función genérica, la firma que coincida para este método y una función para calcular el resultado. Desafortunadamente R no ofrece ningún toque sintáctico para realizar esta tarea por lo que el código es un poco detallado y repetitivo.

setMethod("sides", signature(object = "Polygon"), function(object) {

object@sides

})

Para el resto suministramos valores exactos. Observe que para genéricas con argumentos que usted puede simplificar la firma sin dar los nombres de argumento. Esto ahorra espacio a lo costoso que es el tener que recordar en qué posición corresponde a qué argumento - no es un problema si hay un solo argumento.

setMethod("sides", signature("Triangle"), function(object) 3)

setMethod("sides", signature("Square"), function(object) 4)

setMethod("sides", signature("Circle"), function(object) Inf)

Opcionalmente puede también especificar **valueClass** para definir el resultado esperado de la genérica. Esto producirá un error en tiempo de ejecución si el método devuelve una salida de la clase equivocada.

setGeneric("sides", valueClass = "numeric", function(object) {

standardGeneric("sides")

})

setMethod("sides", signature("Triangle"), function(object) "three")

sides(new("Triangle"))

# invalid value from generic function "sides", class "character", expected

# "numeric"

Tenga en cuenta que los argumentos de que los envíos de carácter genérico sobre no pueden ser evaluados perezosamente - de otra manera, ¿cómo saber qué clase de objeto era R? Esto también significa que se puede utilizar **substitute** para acceder a la expresión sin evaluar.

Para saber qué métodos están ya definidos para una función genérica, utilice **showMethods**:

showMethods("sides")

# Function: sides (package .GlobalEnv) # object="Circle"

# object="Polygon"

# object="Square"

# object="Triangle"

showMethods(class = "Polygon")

# Function: initialize (package methods)

# .Object="Polygon"

# (inherited from: .Object="ANY")

#

# Function: sides (package .GlobalEnv)

# object="Polygon"

#### Métodos de envío

Esta sección describe la estrategia para hacer coincidir una llamada a una función genérica para el método correcto. Si existe una correspondencia exacta entre la clase de los objetos en la llamada, y la firma de un método, es fácil - la función genérica sólo llama a ese método. De lo contrario, R se darán cuenta el método que utiliza el método siguiente:

* Para cada argumento de la función, se calcula la distancia entre la clase en la clase y la clase en la firma. Si son iguales, la distancia es cero. Si la clase en la firma es uno de los padres de la clase en la llamada, entonces la distancia es 1. Si se trata de un abuelo, 2, y así sucesivamente. Calcule la distancia total por la suma de las distancias individuales.
* Calcula la distancia para cada método. Si hay un método con una menor distancia única, usar eso. De lo contrario, dará una advertencia y llamar a uno de los métodos de emparejamiento que se describen a continuación.

Observe que es posible crear métodos que son ambiguos - es decir que no es claro cuál método genérico debe elegir. En este caso R escoge el método que es el primero alfabéticamente y devolverá un mensaje de advertencia acerca de la situación:

setClass("A") setClass("A1", contains = "A")

setClass("A2", contains = "A1")

setClass("A3", contains = "A2")

setGeneric("foo", function(a, b) standardGeneric("foo"))

setMethod("foo", signature("A1", "A2"), function(a, b) "1-2")

setMethod("foo", signature("A2", "A1"), function(a, b) "2-1")

foo(new("A2"), new("A2"))

# Note: Method with signature "A2#A1" chosen for function "foo",

# target signature "A2#A2". "A1#A2" would also be valid

En general, se debe evitar esta ambigüedad, proporcionando un método más específico:

setMethod("foo", signature("A2", "A2"), function(a, b) "2-2")

foo(new("A2"), new("A2"))

(El cómputo se almacena en caché para esta combinación de clases de manera que no tiene por qué hacerse de nuevo.)

Hay dos clases especiales que se pueden utilizar en la firma: **missing** y **ANY**. **missing** corresponde al caso en el que el argumento no es suministrado, y **ANY** se emplea para establecer los métodos por defecto. **ANY** tiene la menor precedencia posible en el método de emparejamiento.

También se puede utilizar las clases básicas como el **numeric**, **character** y **matrix**. Una matriz de (por ejemplo) caracteres debería tener clase **matrix**.

setGeneric("type", function(x) standardGeneric("type"))

setMethod("type", signature("matrix"), function(x) "matrix")

setMethod("type", signature("character"), function(x) "character")

type(letters)

type(matrix(letters, ncol = 2))

Usted también puede enviar en clases S3 siempre y cuando haya realizado un S4 consciente de ello, llamando a **setOldClass**.

foo <- structure(list(x = 1), class = "foo")

type(foo)

setOldClass("foo")

setMethod("type", signature("foo"), function(x) "foo")

type(foo)

setMethod("+", signature(e1 = "foo", e2 = "numeric"), function(e1, e2) { structure(list(x = e1$x + e2), class = "foo")

})

foo + 3

También es posible enviar el ... en circunstancias especiales. Mire **?dotsMethods** para más detalles.

#### Herencia

Vamos a desarrollar un ejemplo más completo. Se inspira en un ejemplo de la referencia de lenguaje Dylan uno de los lenguajes que inspiraron el sistema de objetos S4. En este ejemplo vamos a desarrollar un modelo simple de inspecciones de vehículos que varían en función del tipo de vehículo (coche o camión) y el tipo de inspector (normal o estado).

En S4, es la **callNextMethod** que (¡sorpresa!) se usa para llamar al método siguiente. Ella averigua qué método llamar al pretender que el método actual no existe, y busca la coincidencia más cercana.

Primero necesitamos configurar las clases: dos tipos de vehículos (coches y camiones), y dos tipos de inspección

setClass("Vehicle")

setClass("Truck", contains = "Vehicle")

setClass("Car", contains = "Vehicle")

setClass("Inspector", representation(name = "character"))

setClass("StateInspector", contains = "Inspector")

A continuación definimos la función genérica para inspeccionar un vehículo. Dispone de dos argumentos: el vehículo se está inspeccionando y la persona que realiza la inspección.

setGeneric("inspect.vehicle", function(v, i) {

standardGeneric("inspect.vehicle")

})

Todo vehículo debe ser revisado para la oxidación mediante todos los inspectores, así que vamos a agregar la primera. Los autos también necesita tener cinturones de seguridad en funcionamiento.

setMethod("inspect.vehicle",

signature(v = "Vehicle", i = "Inspector"),

function(v, i) {

message("Looking for rust")

})

setMethod("inspect.vehicle",

signature(v = "Car", i = "Inspector"),

function(v, i) {

callNextMethod() # perform vehicle inspection

message("Checking seat belts")

})

inspect.vehicle(new("Car"), new("Inspector"))

# Looking for rust

# Checking seat belts

Observe que es el método más específico que es responsable de garantizar que los métodos más genéricos son llamados.

A continuación vamos a añadir métodos para camiones (adjuntos de carga deben estar bien), y la tarea especial que el estado desempeña en los coches: la comprobación del seguro.

setMethod("inspect.vehicle",

signature(v = "Truck", i = "Inspector"),

function(v, i) {

callNextMethod() # perform vehicle inspection

message("Checking cargo attachments")

})

inspect.vehicle(new("Truck"), new("Inspector"))

# Looking for rust

# Checking cargo attachments

setMethod("inspect.vehicle",

signature(v = "Car", i = "StateInspector"),

function(v, i) {

callNextMethod() # perform car inspection

message("Checking insurance")

})

inspect.vehicle(new("Car"), new("StateInspector"))

# Looking for rust

# Checking seat belts

# Checking insurance

Este montaje asegura que cuando un inspector estatal verifica un camión llevan a cabo todos los controles de un inspector regular:

inspect.vehicle(new("Truck"), new("StateInspector"))

# Looking for rust

# Checking cargo attachments

### En su hábitat natural

Para terminar, vamos a ver algo de código S4 en la práctica. El paquete Bioconductor **EBImage** por Oleg Sklyar, Pau Gregoire, Mike Smith y Wolfgang Huber es un buen punto para empezar porque es muy simple. Este tiene sólo una clase, **Image**, que representa una imagen como un array de valores de píxeles.

setClass ("Image",

representation (colormode="integer"),

prototype (colormode=Grayscale),

contains = "array"

)

imageData = function (y) {

if (is(y, 'Image')) y@.Data

else y

}

El autor escribió la función auxiliar **imageData** del subyacente para extraer el objeto S3, el array. Ellos también podrían haber utilizado la función **S3Part** para extraer esto.

Los métodos son usados para definir operaciones numéricas para combinar dos imágenes o una imagen con una constante. Aquí el autor está utilizando el grupo genérico **Ops** que coincidirá con todas las llamadas a +, -, \*, ^, %%, %/%, /, ==, >, <, !=, <=, >=, & y |. La función **callGeneric** pasa en esta llamada al método genérico para las matrices. Finalmente cada método comprueba que el objeto modificado es válido, antes de devolverlo.

setMethod("Ops", signature(e1="Image", e2="Image"),

function(e1, e2) {

e1@.Data=callGeneric(imageData(e1), imageData(e2))

validObject(e1)

return(e1)

}

)

setMethod("Ops", signature(e1="Image", e2="numeric"),

function(e1, e2) {

e1@.Data=callGeneric(imageData(e1), e2)

validObject(e1)

return(e1)

}

)

setMethod("Ops", signature(e1="numeric", e2="Image"),

function(e1, e2) {

e2@.Data=callGeneric(e1, imageData(e2))

validObject(e2)

return(e2)

}

)

El paquete Matrix por Douglas Bates y Maechler Martin es un gran ejemplo de en un marco más complicada. Se ha diseñado para almacenar de manera eficiente y calcular con diferentes tipos de matriz especiales. Como en la versión 0.999375-50 se define 130 clases y 24 funciones genéricas. El paquete está bien escrito, bien documentado y bastante fácil de leer. La documento que acompaña **vignette** da una buena descripción de la estructura del paquete. Lo recomiendo encarecidamente descargar el fuente y luego revisar los siguientes archivos R:

* **AllClass.R**: donde todas las clases están definidas
* **AllGenerics.R**: donde todos los genéricos están definidos
* **Ops.R**:, donde los operadores por parejas están definidos, incluyendo la conversión automática de matrices S3 estándar

La mayor parte del trabajo duro se hace en C para la eficiencia, pero aún así es útil analizar los archivos R para ver cómo el código está desarrollado.

## R5

R cuenta con tres sistemas de orientada a objetos (OO): S3, S4 y R5. Esta página describe el nuevo sistema de referencia basado en clases, coloquialmente conocida como la R5.

R5 es algo nuevo, aparece en R 2.12. Estos llenan una necesidad de hace mucho tiempo para los objetos mutables que previamente habían sido cubiertos por paquetes no centrales como **R.oo**, **proto** y **mutatr**. Mientras que la funcionalidad central de R5 es sólida, las clases de referencia están aún en desarrollo y algunos detalles van a cambiar. La versión más actualizada de la documentación para R5 se la puede encontrar en **?ReferenceClasses**.

Existen dos principales diferencias entre R5 y S3 y S4:

* los objetos R5 utilizan transferencia de mensajes OO
* los objetos R5 son mutables: la usual copia de R que modifica la semántica no es aplicable

Estas propiedades hacen de este sistema de objetos comportarse más como Java y C #. Sorprendentemente, la implementación de R5 esta casi en su totalidad en código R - ellos son una combinación de métodos y entornos. Este es un testamento para la flexibilidad de S4.

Particularmente indicados para: simulaciones donde usted está modelado situaciones complejas, interfaces gráficas de usuario.

Observe que cuando se usa referencias basadas en clases queremos minimizar los efectos colaterales, y utilizarlos sólo cuando el estado mutable es requerido. La mayoría de las funciones aún deben ser "funcionales", y libres de efectos colaterales. Esto hace el código fácil de comprender (porque usted no necesita preocuparse sobre los métodos cambiantes de manera sorprendente), y es más fácil de entender para otros programadores R.

Limitaciones: No se puede utilizar un entorno cerrado - porque este es usado por el objeto.

### Clases e instancias

Crear nueva una referencia basado en clases es sencillo: se utiliza **setRefClass**. A diferencia de **setClass** de S4, se quiere mantener los resultados de la función, porque esto es lo que se usara para crear nuevos objetos de ese tipo:

# Or keep reference to class around.

Person <- setRefClass("Person")

Person$new()

Una referencia de clase tiene tres componentes principales, dados por tres argumentos para **setRefClass**:

* **contains**, las clases que heredo de la clase. Estos deberían ser otras referencias a clases de objetos:

setRefClass("Polygon")

setRefClass("Regular")

# Specify parent classes

setRefClass("Triangle", contains = "Polygon")

setRefClass("EquilateralTriangle",

contains = c("Triangle", "Regular"))

* **fields** son el equivalente de slots en S4. Ellos pueden ser especificados como un vector de nombres de campos, o una lista llamada de tipos de campos:

setRefClass("Polygon", fields = c("sides"))

setRefClass("Polygon", fields = list(sides = "numeric"))

La propiedad más importante de objetos R5 es que ellos son mutables, o equivalentemente ellos tienen referencia semántica:

Polygon <- setRefClass("Polygon", fields = c("sides"))

square <- Polygon$new(sides = 4)

triangle <- square

triangle$sides <- 3

square$sides

* **methods** son funciones que operan dentro del contexto del objeto y puede modificar sus campos. Estos también se pueden añadir después de la creación de objetos, como se describe a continuación.

setRefClass("Dist")

setRefClass("DistUniform", c("a", "b"), "Dist", methods = list(

mean <- function() {

(a + b) / 2

}

))

Usted también puede agregar métodos después de la creación:

# Instead of creating a class all at once:

Person <- setRefClass("Person", methods = list(

say\_hello = function() message("Hi!")

))

# You can build it up piece-by-piece

Person <- setRefClass("Person")

Person$methods(say\_hello = function() message("Hi!"))

Actualmente no es posible modificar los campos porque la adición de campos invalidaría los objetos existentes que no contaban con esos campos.

El objeto devuelto por **setRefClass** (o recuperada más tarde por **getRefClass**) se llama un objeto generador. Este dispone de métodos:

**new** para crear nuevos objetos de esa clase. El nuevo método tiene argumentos con nombres especificando los valores iniciales para los campos

* **methods** para modificar existentes o añadir nuevos métodos
* **help** para obtener ayuda acerca de los métodos
* **fields** para obtener una lista de campos definidos para la clase
* **lock** bloquea los campos con nombre de manera que su valor sólo puede establecerse una vez
* **accessors** un método conveniente que establece automáticamente la forma de acceso de **getXXX** y **setXXX**.

### Métodos

los métodos R5 están asociados con objetos, no con funciones y son llamados utilizando la sintaxis especial **obj$method (arg1, arg2, ...)**. (Usted podría recordar que hemos visto antes de esta construcción cuando llamamos funciones almacenadas en una lista nombrada). Los métodos también son especiales, ya que ellos pueden modificar los campos. Esto es diferente

También hemos visto que antes de esta construcción, cuando usamos el cierre para crear un estado mutable. Las clases de referencia trabajan de manera similar, pero estas nos dan alguna funcionalidad extra:

* herencia
* una manera de documentar métodos
* una manera de especificar campos y sus tipos

Modificar campos con **<<-**. Llamará funciones de acceso si se ha definido.

Campos especiales: **.self** (no use campos con nombres que comienzan con **.** que estos pueden ser usados para propósitos especiales en futuras versiones.) **initialize**

#### Métodos comunes

Debido a que todas las clases R5 heredan de la misma superclase **envRefClass**, ellos tienen un conjunto común de métodos:

* **obj$callSuper**:
* **obj$copy**: crea una copia del objeto actual. Esto es necesario ya que las clases R5 no se comportan como la mayoría de objetos R, las cuales se copian sobre la asignación o modificación.
* **obj$field**: denominado acceso a los campos. Equivalente a **slots** para S4. **obj$field**("xxx") el mismo que **obj$xxx**. **obj$field**("xxx", 5) el mismo que **obj$xxx <- 5**
* **obj$import**(x) coacciona en este objeto y **obj$export**(Class) coacciona a una copia de obj en esa clase. Estas deben ser superclases.
* **obj$initFields**

#### Documentación

Python estilo doc-strings. obj$help().

### En paquetes

Nota: collation Nota: namespaces y exporting

### En su habitat natural

Rook paquete. Scales package?

# SoftwareSystems

En el nivel más básico, un programa R, al igual que cualquier otro programa es una secuencia de instrucciones escritas para realizar una tarea. Los programas constan de estructuras de datos que albergan datos y funciones, que definen cosas qué que un programa puede hacer. Usted ya está familiarizado con las estructuras de datos nativas de R: vectores, listas, data frames , etc Y ya ha visto las funciones que acceden y manipular estas funciones. Sin embargo, al diseñar sus propios sistemas sobre de R es muy probable que desee crear sus propias estructuras de datos. Después de que estos nuevos tipos están definidos es posible que desee crear funciones especializadas que operen en sus nuevas estructuras de datos. En otros casos es posible que desee extender los sistemas existentes para sacar provecho de su nueva funcionalidad. Este capítulo muestra cómo construir nuevos sistemas de software que puede "se conectan a" la funcionalidad existente de R y permite a otros usuarios ampliar sus nuevas capacidades .

Las estructuras de datos están generalmente asociados con un conjunto de funciones creadas para trabajar con ellos. Las estructuras de datos y sus funciones pueden ser encapsulados para crear clases. Las clases nos ayudan a compartimentar piezas conceptualmente coherentes de software. Por ejemplo, un vector de R es la clase que sostiene un secuencia de tipos atómicos en R. Podemos crear una instancia de un vector usando una de las rutinas de R de creación de vectores.

x <- 1:10

length(x)

La variable **x** es un objeto de tipo vector. Cuando la clase describe qué la estructura de datos se verá como un objeto es una instancia real de ese tipo. Los objetos son asociados con funciones que nos permiten hacer cosas como el acceso y manipular los datos en posesión de un objeto. En el ejemplo anterior, la función length está asociada con vectores y nos permite conocer cuántos elementos del vector se mantiene.

R proporciona tres construcciones diferentes para programar con clases, llamadas también orientada a objetos (OO), **S3, S4, y R5**. Los dos primeros S3 y S4 están escritas en un estilo llamado funciones genéricas OO. Las funciones que pueden asociarse con una clase se define primero como funciones genéricas. Entonces métodos o funciones asociados con una clase específica, se definen como cualquier otra función. Sin embargo, cuando una instancia de un objeto se pasa a la función genérica como parámetro, se envía a su método asociado. R5 es implementado en un estilo denominado transferencia de mensajes OO. En este estilo unos métodos están directamente asociados con las clases y es el objeto que determina qué función llamar.

Para el resto de este capítulo vamos a explorar el uso de S3, S4, R5 y para generar secuencias. Junto con la construcción de un sistema general de generación de secuencias nosotros vamos a crear clases que generan los números de Fibonacci, uno por uno. Como usted probablemente ya sabe, los números de Fibonacci siguen la secuencia de enteros

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144 ...

y se definen por recurrencia

F (0) = 0 F (1) = 1 F (k) = F (k-1) + F (k-2)

Estos números pueden ser fácilmente generados en R utilizando los habituales vectores y funciones que usted ya conoce. Un ejemplo de cómo hacer esto es proporcionado a continuación. Es importante comprender que las técnicas mostradas en este capítulo no permiten expresar algoritmos tu no puedes expresar con las estructuras nativas de R datos y funciones. Las técnicas que le permiten organizar las estructuras de datos y funciones para crear un sistema general o marco para generar secuencias.

fibonacci <- function(lastTwo=c()) {

if (length(lastTwo) == 0) {

lastTwo <- 1

} else {

lastTwo <- c(lastTwo, sum(lastTwo))

if (length(lastTwo) > 2) {

lastTwo <- lastTwo[-1]

}

}

return(lastTwo)

}

# Get the first 10 fibonacci numbers

fibs <- fibonacci()

for (i in 1:10) {

print(tail(fibs, 1))

fibs <- fibonacci(fibs)

}

La creación de un sistema general para secuencias tiene dos ventajas. En primer lugar, permite la abstracción. En nuestro ejemplo hemos definido un vector para mantener los dos últimos valores de la sucesión de Fibonacci, junto con una función que obtiene el valor siguiente según el orden. Al darnos cuenta que cualquier secuencia de enteros que nos gustaría generar computacionalmente puede ser expresada como datos, los dos últimos valores para la secuencia Fibonacci, y una función para obtener el siguiente valor. Hemos identificado las piezas esenciales de generación de secuencias. A partir de aquí podemos empezar a pensar en el tipo de cosas que nos gustaría hacer con cualquier otra secuencia, no sólo los Fibonaccis. En segundo lugar, podemos hacer extensible nuestro sistema. Es decir, podemos escribir código para otros tipos de secuencias que trabajan dentro de nuestro marco. La extensibilidad le permite crear nuevas secuencias, como los números factoriales, basado en la noción abstracta de secuencia. Incluso permitirá a otros definir sus propias secuencias que funcionen dentro de nuestro marco de secuencia.

# Cómo seguir con éxito convenciones de nomenclatura en R

R es muy liberal cuando se trata de nombres para objetos y funciones. Esta libertad es una gran bendición y una gran carga al mismo tiempo. Nadie está obligado a seguir las reglas estrictas, por lo que todos los que algo programan en R, básicamente, puede hacer lo que quieran.

## Cómo elegir un nombre correcto en R

Aunque casi todo está permitido al dar nombres a los objetos, todavía hay algunas reglas en R que no se puede ignorar:

* **Los nombres deben comenzar con una letra o un punto.** Si inicia un nombre con un punto, el segundo carácter no puede ser un dígito.
* **Los nombres deben contener sólo letras, números, guiones bajos (\_) y puntos (.).** Aunque puede forzar a R para aceptar otros caracteres en los nombres, no debería, porque estos caracteres suelen tener un significado especial en R.
* **No se pueden utilizar las siguientes palabras clave especiales como los nombres:**
  + break
  + else
  + FALSE
  + for
  + function
  + if
  + Inf
  + NA
  + NaN
  + next
  + repeat
  + return
  + TRUE
  + while

R es sensible a mayúsculas y minúsculas, lo que significa que, para R, lastname  y Lastname  son dos objetos diferentes. Si R le dice que no puede encontrar un objeto o una función y que está seguro de que debería estar allí, compruebe que está utilizando el caso correcto.

## Elegir nombres en R que añaden significado a su código

Cuando Joris era joven, sus padres compraron un cordero lindo que necesitaba un nombre. Después de mucha contemplación, decidió llamarlo Blacky. No importaba que el cordero era realmente blanco y su nombre hizo que todos los demás creyeran que se trataba de un perro, Joris pensó que era un nombre perfecto.

Asimismo, llamando al resultado de una secuencia de comandos larga Blacky  puede ser un poco confusa para la persona que tiene que leer el código más adelante, incluso si tiene todo tipo de sentido para usted. ***Recuerde***: Usted podría ser el que, en tres meses, está tratando de averiguar exactamente lo que estaba tratando de lograr. El uso de nombres descriptivos le permitirá mantener el código legible.

Aunque se puede nombrar un objeto de la manera que quiera, algunos nombres causarán menos problemas que otros. Se habrá dado cuenta de que ninguna de las funciones que hemos utilizado hasta ahora es mencionada como fuera de límites. Así es: Si desea llamar a un objeto paste, es libre de hacerlo:

> paste <- paste("This gets","confusing")

> paste

[1] "This gets confusing"

> paste("Don't","you","think?")

[1] "Don't you think?"

R siempre sabrá perfectamente bien cuando desea el vector paste  y cuando se necesita la función paste(). Eso no quiere decir que sea una buena idea usar el mismo nombre para ambos elementos, sin embargo. Si usted puede evitar dar el nombre de una función a un objeto, debería hacerlo.

Una situación en la que realmente se puede meter en problemas es cuando se utiliza mayúsculas F  o T como un nombre de objeto. Se puede hacer, pero es muy probable romper el código en algún momento. Aunque es una idea muy mala, T y F  se utilizan con demasiada frecuencia como abreviaturas para TRUE  y FALSE, respectivamente. Pero T y F no son palabras clave reservadas.

Así que, si usted los cambia, R buscará primero el objeto T  y sólo entonces tratará de sustituir con T  TRUE. Y cualquier código que aún espera T  para significar TRUE  producirá un error de ahora en adelante. Nunca utilice F o T, no como un nombre de objeto y no como una abreviatura.

# Cómo depurar el código en R

Una vez que haya escrito el código en R, es importante saber cómo depurarlo. Errar es humano, y los programadores también caen en esa categoría "humana". Nadie se las arregla para escribir código sin errores, así que en vez de preguntarse *si* tiene errores en su código, usted debe preguntarse *donde* tiene errores en su código.

## Conozca lo que debe buscar en R

Un *bug* (insecto) es simplemente otra palabra para algún tipo de error en su programa. Por lo tanto, *debugging*  no implica insecticidas — simplemente significa deshacerse de todo tipo de errores semánticos y / o lógicos en sus funciones.

Antes de comenzar la caza de bugs, usted tiene que saber lo que estás buscando. En general, se puede dividir los errores en el código en tres categorías diferentes:

* **Errores de sintaxis:** Si escribe código que R no puede entender, tiene errores de sintaxis. Los errores de sintaxis siempre resultan en un mensaje de error y, a menudo son causados ​​por la falta de ortografía de una función u olvidar un paréntesis.
* **Errores semánticos:** Si usted escribe código correcto que no hace lo que usted cree, usted tiene un error semántico. El propio código es correcto, pero el resultado de esa línea de código no lo es. Puede, por ejemplo, devolver otro tipo de objeto al que usted espera. Si utiliza el objeto más adelante, no va a ser el tipo que piensa que es y su código producirá un error.
* **Errores lógicos:** Probablemente los más difíciles de encontrar son los errores en la lógica del código. Su código funciona, lo que no genera ningún error o advertencia, pero todavía no devuelve el resultado que usted espera. El error no está en el código en sí, sino en la lógica ejecutada.

Esto puede parecer un pequeño detalle, pero encontrar diferentes tipos de errores requiere de estrategias diferentes. A menudo, usted puede localizar fácilmente un error de sintaxis con la simple lectura de los mensajes de error, pero los errores semánticos representan un desafío completamente diferente y los errores lógicos pueden ocultarse en el código sin ser consciente de su existencia.

# Cómo decirle a R la función para Debug

Usted puede caminar a través de una función y después decirle a R que desea depurar utilizando la función debug(), así:

> debug(logit)

De ahora en adelante, R cambiará al modo de navegador cada vez que la función se llama desde cualquier lugar en R, hasta que le indique a R explícitamente para detener la depuración o hasta que se sobrescribe la función a través de outsourcing de nuevo. Para detener la depuración de una función, basta con utilizar undebug(logit).

Si desea desplazarse por una función una sola vez, puede utilizar la función debugonce()en lugar de debug(). R irá a modo de navegador la próxima vez que se llame a la función, y sólo entonces — por lo que no es necesario utilizar undebug()para detener la depuración.

Si intenta con la función logitpercent()de nuevo después de ejecutar el código debug(logit), verá lo siguiente:

> logitpercent('50%')

debugging in: logit(as.numeric(x))

debug at D:/RForDummies/Ch10/logitfunc.R#2: {

x <- ifelse(x < 0 | x > 1, "NA", x)

log(x/(1 - x))

}

Browse[2]>

Usted ve que el prompt cambia. Ahora dice Browse[2]. Este mensaje le indica que está navegando dentro de una función.

El número indica el nivel del apilamiento de llamada que está visitando en ese momento. Recuerde, de la salida de la función traceback() , que la función logit() ocurre como la segunda función del apilamiento de llamada. Ese es el número 2 en la salida anterior.

El texto adicional sobre el prompt modificado le brinda la siguiente información:

* La línea desde donde se llama a la función — en este caso, la línea logit(as.numeric(x))de la función logitpercent()
* El archivo o la función que se depura — en este caso, el archivo logitfunc.R, empezando en la línea segunda
* Parte del código que está a punto de navegar a través de

# Cómo paso un Debugging sobre funciones en R

Usted puede caminar a través de una función después de decirle a R que desea depurar utilizando la función debug(). A partir de entonces, R cambiará al modo de navegador cada vez que la función se llama desde cualquier lugar de R, hasta que le indique a R explícitamente para detener la depuración o hasta que se sobrescriba la función a través del outsourcing de nuevo.

Para activar el paso a paso de la depuración de la función, utilice debug(logit)

Para detener la depuración de una función, basta con utilizar undebug(logit).

Si desea desplazarse por una función de una sola vez, puede utilizar la función debugonce() en lugar de debug(). R irá a modo de navegador la próxima vez que se llame a la función, y sólo entonces — por lo que no es necesario utilizar undebug() para detener la depuración.

Si intenta con la función logitpercent()de nuevo después de ejecutar el código debug(logit), verá lo siguiente:

> logitpercent('50%')

debugging in: logit(as.numeric(x))

debug at D:/RForDummies/Ch10/logitfunc.R#2: {

x <- ifelse(x < 0 | x > 1, "NA", x)

log(x/(1 - x))

}

Browse[2]>

Usted ve que el prompt cambia. Ahora dice Browse[2]. Este mensaje le indica que está navegando dentro de una función.

El número indica qué nivel del apilamiento de llamada está visitando en ese momento. Recuerde, de la salida de la función traceback()que la función logit() ocurre como la segunda función en el apilamiento de llamada. Ese es el número 2 en la salida anterior.

El texto adicional sobre el prompt modificado contiene la siguiente información:

* La línea desde donde se llama a la función — en este caso, la línea logit(as.numeric(x))de la función logitpercent()
* El archivo o la función que se depura — en este caso, el archivo logitfunc.R, empezando en la línea segunda
* Parte del código que está a punto de navegar a través de

# Cómo cazar Bugs en R

La caza de Bugs en R a veces puede ser una situación difícil. Aunque el mensaje de error siempre le dice qué línea de código genera el error, no puede ser la línea de código donde las cosas empezaron a ir mal. Esto hace la caza de un bug un negocio complejo, pero algunas estrategias simples pueden ayudarle a localizar a estas criaturas molestas.

## Calcular el logit

Para ilustrar algunas de las estrategias de caza de bugs en R, use un ejemplo sencillo. Digamos, por ejemplo, su colega escribió dos funciones para calcular el logit para ambas proporciones y porcentajes, pero no puede conseguir que funcione. Por lo tanto, se le pide que ayude a encontrar bugs. Aquí está el código que le envía:

# checks input and does logit calculation

logit <- function(x){

x <- ifelse(x < 0 | x > 1, "NA", x)

log(x / (1 - x) )

}

# transforms percentage to number and calls logit

logitpercent <- function(x){

x <- gsub("%", "", x)

logit(as.numeric(x))

}

Copie y pegue este código en el editor y guarde el archivo utilizando, por ejemplo, logitfunc.R  como su nombre. Después de eso, origine el archivo en el editor de R utilizando la función source()o el botón fuente o el comando del editor de su elección. Ahora el código de función se carga en R, y ya está listo para comenzar la caza.

El logit no es otra cosa que el logaritmo de las probabilidades, calculado como log(x / (1-x))si x  es la probabilidad de un acontecimiento que tiene lugar. Los estadísticos utilizan esto a la hora de modelar los datos binarios mediante modelos lineales generalizados. Si alguna vez tiene que calcular un logit usted mismo, usted puede utilizar la función qlogis()para eso. Para calcular las probabilidades de los valores logit, se utiliza la función plogis().

## Saber de dónde proviene un error

Su compañero se quejó de que él tiene un error al intentar el siguiente código:

> logitpercent('50%')

Error in 1 - x : non-numeric argument to binary operator

Efectivamente, pero no encuentra el código 1 - x  en el cuerpo de logitpercent(). Por lo tanto, el error viene de otro lugar. Para saber de dónde, se puede utilizar la función traceback() inmediatamente después de que se produjo el error, así:

> traceback()

2: logit(as.numeric(x)) at logitfunc.R#9

1: logitpercent("50%")

Esta función traceback() imprime lo que se denomina el *call stack*  (apilamiento de llamada) que conduce al error anterior. Esta pila de llamadas representa la secuencia de llamadas de función, pero en orden inverso. La función de la parte superior es la función en la que se genera el error real.

En este ejemplo, R llama a la función logitpercent() , y la función que, a su vez, llama a logit(). El traceback le dice que el error se produjo dentro de la función logit(). Aún más, la función traceback() le indica que se produjo el error en la línea 9 del archivo de código logitfunc.R , como se indica por logitfunc.R#9 en la salida de traceback().

La pila de llamadas le da un montón de información — a veces demasiado. Se puede apuntar a una oscura función interna como la que arrojó el error. Si dicha función no suena, compruebe superiormente en la pila de llamadas de una función a reconocer y empiece a depurar desde allí.

# Cómo saber cuándo cuidar de las advertencias en R

Los errores y advertencias tienen diferentes funciones en R. Usted no puede moverse alrededor de los errores, porque simplemente paran su código. Advertencias en otro lado son una cosa completamente diferente. Incluso si R lanza una advertencia, se continúa ejecutando el código independientemente. Puede pasar por alto las advertencias, pero en general es una idea muy mala. Las advertencias a menudo son el único signo que tiene de que el código tiene algún error de semántica o lógica.

Por ejemplo, podría haber olvidado el ifelse() y se trató de algo como el siguiente ejemplo:

> x <- 1:10

> y <- if (x < 5 ) 0 else 1

Warning message:

In if (x < 5) 0 else 1 :

the condition has length > 1 and only the first element will be used

Esta advertencia apunta a un error semántico: if  espera un valor único TRUE o FALSE , pero proporciona un vector conjunto. Tenga en cuenta que, al igual que los errores, las advertencias dicen en general que el código ha generado el aviso.

He aquí otra advertencia que aparece con regularidad y puede apuntar a un error de semántica o lógica en el código:

> x <- 4

> sqrt(x - 5)

[1] NaN

Warning message:

In sqrt(x - 5) : NaNs produced

Debido a que x - 5 es negativo cuando x  es 4, R no puede calcular la raíz cuadrada y le advierte que la raíz cuadrada de un número negativo no es un número (NaN).

Si es un matemático, es posible señalar que la raíz cuadrada de –1 es 0 - 1i. R puede, de hecho, hacer cálculos con números complejos, pero entonces usted tiene que definir las variables como números complejos. Se puede comprobar, por ejemplo, el archivo de ayuda ?complex para más información.

Aunque la mayoría de las advertencias, ya sea como resultado de errores de semántica o lógica en el código, incluso un error de sintaxis simple puede generar una advertencia en lugar de un error. Si desea trazar algunos puntos en R, se utiliza la función plot(). Se necesita un argumento col  para especificar el color de los puntos, pero por error se podría tratar de colorear los puntos utilizando lo siguiente:

> plot(1:10, 10:1, color='green')

Si intenta esto, usted consigue seis mensajes de alerta a la vez, todo lo que está diciendo es que color  no es probablemente el nombre del argumento que estaba esperando:

Warning messages:

1: In plot.window(...) : "color" is not a graphical parameter

2: In plot.xy(xy, type, ...) : "color" is not a graphical parameter

....

Tenga en cuenta que los mensajes de advertencia no apuntan hacia el código que se escribe en la línea de comandos, sino que apuntan a las funciones que nunca antes se usaron, como plot.window() y plot.xy().

Puede pasar argumentos de una función a otra con el argumento de puntos. Eso es exactamente lo que plot() hace aquí. Por lo tanto, plot()no genera una advertencia, pero todas las funciones que plot() pasan el argumento color lo hacen.

Si recibe mensajes de advertencia o error, una mirada a fondo en las páginas de ayuda de la(s) función (es) que ha generado el error puede ayudar a determinar cuál es la razón para que el mensaje que recibió. Por ejemplo, en la página de ayuda de ?plot.xy, se encuentra que el nombre correcto del argumento es col.

Así que, para resumir, la mayoría de las advertencias apuntan a uno de los siguientes problemas:

* La función le dio un resultado, pero por alguna razón, ese resultado no puede ser correcto.
* La función genera un resultado atípico, al igual que los valores de NA o NaN.
* La función no puede hacer frente a algunos de los argumentos y los ignora.

Sólo el último le dice que hay un problema con su sintaxis. Para los demás, hay que examinar el código un poco más.

# Cómo leer errores y advertencias en R

Si algo va mal con su código, R le explica. Tenemos que admitirlo: Estos mensajes de error pueden ir desde una ligera confusión a completamente incomprensible si no se está acostumbrado a ellos. Pero no tiene que ser de esa manera. Cuando usted se familiarice con el error y mensajes de advertencia de R, usted puede advertir lo que está mal. Los errores pueden ocurrir de dos maneras:

* El código sigue funcionando hasta el final, y cuando el código está terminado, R imprime un mensaje de advertencia.
* El código se detiene inmediatamente porque R no puede llevarlo a cabo, y R imprime un mensaje de error.

**Leer mensajes de error**

Echemos un vistazo a ese mensaje de error. Si se intenta el siguiente código, se recibe este, más o menos claro, mensaje de error:

> "a" + 1

Error in "a" + 1 : non-numeric argument to binary operator

Usted recibe dos bits de información en este mensaje de error. En primer lugar, la línea "a" + 1 le indica en qué línea de código se tiene un error. Adicionalmente, le dice cuál es el error. En este caso, se utilizó un argumento no numérico (el caracter 'a') en combinación con un operador binario (el signo +).

R siempre le dice en qué parte del código se produce el error, para que sepa que en muchos casos donde hay que empezar a buscar.

Los mensajes de error no siempre son tan claros. Observe el siguiente ejemplo:

> data.frame(1:10,10:1,)

Error in data.frame(1:10, 10:1, ) : argument is missing, with no default

¿A qué argumentos se refieren este error? En realidad, se refiere a un argumento vacío que se ha facilitado para la función. Después de que el segundo vector, hay una coma que no debería estar allí. Un pequeño error de escritura, pero R espera otro argumento después de la coma y no encuentra uno.

Si usted no entiende inmediatamente un mensaje de error, tome una mirada más cercana a las cosas que el mensaje de error está explicando. En ocasiones usted puede escribir algo mal.

**Cómo reconocer y corregir Lista de Errores en R**

Aunque las listas ayudan con el mantenimiento de los datos y pueden ser muy útiles cuando se está procesando múltiples bases de datos en R, también pueden causar algunos problemas.

En primer lugar, es fácil olvidar que ciertas funciones devuelven una lista en lugar de un vector. Por ejemplo, muchos programadores olvidan que strsplit() devuelve una lista en lugar de un vector. Por lo tanto, si desea que la segunda palabra de una frase, el siguiente código no devuelve un error, pero no da la respuesta correcta:

> strsplit('this is a sentence',' ')[2]

[[1]]

NULL

En este ejemplo, strsplit() devuelve una lista con un elemento, el vector con las palabras de la oración:

> strsplit('this is a sentence',' ')

[[1]]

[1] "this" "is" "a" "sentence"

Para acceder a este vector, primero se tiene que seleccionar el elemento deseado de la lista. Sólo entonces se puede buscar el segundo valor utilizando los índices del vector, así:

> strsplit('this is a sentence',' ')[[1]][2]

[1] "is"

Incluso, el mecanismo de indexación en sí mismo puede causar errores de este tipo. Por ejemplo, usted tiene algunos nombres de clientes y desea agregar un punto entre sus nombres y apellidos. De este modo, en primer lugar, se los divide así:

> customer <- c('Johan Delong','Marie Petit')

> namesplit <- strsplit(customer,' ')

Usted desea pegar el segundo nombre junto con un punto en el medio, por lo que necesita seleccionar el segundo elemento de la lista. Si utiliza corchetes simples, se obtiene lo siguiente:

> paste(namesplit[2],collapse='.')

[1] "c(\"Marie\", \"Petit\")"

Eso no es lo que usted desea en absoluto. Recuerde que usted puede utilizar ambos, corchetes simples y corchetes dobles para seleccionar elementos de una lista, pero cuando se utiliza corchetes simples, siempre obtendrá una lista. Por lo tanto, para obtener el resultado correcto, necesita corchetes dobles, por ejemplo:

> paste(namesplit[[2]],collapse='.')

[1] "Marie.Petit"

Note R nunca dio una señal - ni siquiera una advertencia – de que algo andaba mal. Por lo tanto, si usted observa listas que no espera (o no las encuentra en el que usted esperaba), revise sus corchetes.

**Cómo corregir los errores de lectura de datos en R**

Es probable que los errores más comunes en R se realicen durante la lectura de los datos de archivos de texto usando read.table() o read.csv(). Muchos errores dan lugar a errores de lanzamiento en R, pero a veces sólo se nota que algo salió mal cuando nos fijamos en la estructura de los datos. En este último caso, a menudo se encuentra que las algunas o todas las variables se convierten en factores cuando en realidad no debería ser así.

Cuando R da errores o la estructura de los datos no es lo que usted piensa que debería ser, compruebe lo siguiente:

* **¿Se le olvidó especificar el argumento** header = TRUE**?** Si es así, R verá los nombres de columna como valores y, como resultado, convertirá cada variable a un factor como lo hace siempre con datos de caracteres en un archivo de texto.
* **¿Tuvo espacios en los nombres de columna o datos?** La función read.table() puede interpretar los espacios en, por ejemplo, nombres de columna o en datos de cadena como un separador. Se obtiene errores diciendo 'line x did not have y elements'.
* **¿Tuvo un separador decimal diferente**? En algunos países, los decimales se separan con una coma. Tiene que decirle específicamente a R cuál es el caso con el argumento dec = "," en la función theread.table().
* **¿Se le olvidó especificar** stringsAsFactors = FALSE**?** De forma predeterminada, R cambia los datos de caracteres de los factores, por lo que siempre hay que añadir este argumento si desea que sus datos permanezcan como variables de caracteres.
* **¿Tiene otra forma de especificar los valores perdidos?** R lee 'NA' en un archivo de texto como un valor perdido, pero el archivo puede utilizar un código diferente (por ejemplo, 'missing'). R lo verá como texto de nuevo y convertirá esa variable a un factor. Puede resolver esto especificando el argumento na.strings en ​​la función read.table().

Si siempre comprueba la estructura de sus datos inmediatamente después de haberlos leído, puede detectar errores mucho antes y evitar horas de frustración. Su mejor opción es utilizar str() para obtener información sobre los tipos y head() para ver si los valores son los correctos.

# Cómo generar sus propios mensajes de error en R

La generación de sus propios mensajes puede sonar extraño, pero en realidad se puede evitar resbalones en R mediante la generación de sus propios errores. ¿Recuerda el error lógico en la función logitpercent()? Hubiera sido más fácil de detectar si la función logit() devolvía un error que indica que ha aprobado un número mayor que 1.

Añadiendo mensajes sensibles de error (o advertencias) a una función, puede ayudar a depurar funciones futuras donde usted llama a esa función específica de nuevo. Es de especial utilidad en la búsqueda de errores semánticos o lógicos que sean de otra manera difícil de encontrar.

## Cómo crear mensajes de error en R

Le puede decir a R que emita un error mediante la inserción de la función stop() en cualquier parte del cuerpo de la función, como en el ejemplo siguiente:

logit <- function(x){

if( any(x < 0 | x > 1) ) stop('x not between 0 and 1')

log(x / (1 - x) )

}

Con la sentencia if(), se prueba si algún valor de x está entre 0 y 1. Usando la función any() en torno a la condición se permite al código trabajar con vectores completos, en lugar de los valores individuales. Debido a que la función log() funciona vectorizado, de este modo, toda la función está vectorizada.

Si cambia el cuerpo de la función logit(), de esta manera, y tratar de calcular el logit de 50% y el 150% (o 0,5 y 1,5), R lanza un error como el siguiente:

> logitpercent(c('50%','150%'))

Error in logit(as.numeric(x)/100) : x not between 0 and 1

Como el nombre indica, la ejecución del código en cualquier momento se detiene con la función stop() que se lleva a cabo efectivamente, por lo que no devuelve ningún resultado.

## Cómo crear mensajes de advertencia en R

También puede hacer la función de generar una advertencia en lugar de un error. De esta manera usted obtiene la misma información, pero la función completa se lleva a cabo para que pueda obtener también un resultado.

Para generar una advertencia, utilice la función warning() en lugar de stop(). Por lo tanto, para obtener el resultado deseado, sólo tiene que cambiar el cuerpo de la función al siguiente código:

x <- ifelse(x < 0 | x > 1, NA, x )

if( any(is.na(x)) ) warning('x not between 0 and 1')

log(x / (1 - x) )

Si usted prueba la función ahora, se obtiene el resultado deseado:

> logitpercent(c('50%','150%'))

[1] 0 NA

Warning message:

In logit(as.numeric(x)/100) : x not between 0 and 1

No sólo hace que la función NA retorne cuando se debe, sino que también le da una advertencia que puede ayudar con la depuración de las demás funciones que utilizan la función logit() en algún lugar del cuerpo del código.

**Cómo utilizar Vectorización con sentencias if en R**

Vectorización es uno de los atributos determinantes del lenguaje R. R no sería R si no tuviera algún tipo de versión vectorizada de una instrucción if... else.

## El problema

La función priceCalculator() todavía no es muy económica en su uso. Si usted tiene 100 clientes, tendrá que calcular el precio de cada cliente por separado. Compruebe por usted mismo lo que sucede si se agrega, por ejemplo, tres diferentes cantidades de horas como argumento:

> priceCalculator(c(25,110))

[1] 1060 4664

Warning message:

In if (hours > 100) net.price <- net.price \* 0.9 :

the condition has length > 1 and only the first element will be used

No sólo R advierte que algo raro está pasando, sino que el resultado que se obtiene es simplemente erróneo. En lugar de $ 4.664, el segundo cliente debe cobrar sólo $ 4.198:

> priceCalculator(110)

[1] 4198

El mensaje de advertencia le debe dar una idea clara acerca de lo que pasó. Una sentencia if se puede tratar sólo con un único valor, pero la expresión hours > 100  devuelve dos valores, como se muestra en el siguiente código:

> c(25, 110) > 100

[1] FALSE TRUE

## Elegir basado en un vector lógico en R

La solución que estamos buscando es la función ifelse(), que es una forma vectorizada de elegir valores de dos vectores. Esta función notable toma tres argumentos:

* Un vector test con valores lógicos
* Un vector con los valores que se deben devolver si el valor correspondiente en el vector test es TRUE
* Un vector con los valores que se deben devolver si el valor correspondiente en el vector test es FALSE

### Cómo funciona

Eche un vistazo al siguiente ejemplo:

> ifelse(c(1,3) < 2.5 , 1:2 , 3:4)

[1] 1 4

1. **La expresión condicional c(1,3)<2,5 se evalúa a un vector lógico.**
2. **El primer valor de este vector es TRUE, porque 1 es menor que 2,5. Así, el primer valor del resultado es el primer valor del segundo argumento, que es 1.**
3. **El siguiente valor es FALSE, porque 3 es mayor que 2,5. Por lo tanto, ifelse() toma el valor de la segundo valor del tercer argumento (que es 4) como el segundo valor del resultado.**
4. **Un vector con los valores seleccionados se devuelve como el resultado.**

### Prúebelo

Para ver cómo funciona esto en el ejemplo de la función priceCalculator(), pruebe la función de salida en la línea de comando en la consola. Digamos que usted tiene dos clientes y trabajó 25 y 110 horas para ellos, respectivamente. Usted puede calcular el precio neto con el siguiente código:

> my.hours <- c(25,110)

> my.hours \* 40 \* ifelse(my.hours > 100, 0.9, 1)

[1] 1000 3960

Recuerde que la función ifelse() puede reciclar sus argumentos. Y eso es exactamente lo que hace aquí. En la anterior llamada a la función ifelse(), puede traducir el vector lógico creado por la expresión my.hours> 100 en un vector que contiene los números 0,9 y 1 en lugar de TRUE y FALSE, respectivamente.

**Adaptar la función en R**

Por supuesto, es necesario adaptar la función priceCalculator() en tal forma que también pueda introducir un vector con los valores para el argumento public. De lo contrario, no sería capaz de calcular los precios de una mezcla de clientes públicos y privados. La función final es la siguiente:

priceCalculator <- function(hours,pph=40,public){

net.price <- hours \* pph

net.price <- net.price \* ifelse(hours > 100 , 0.9, 1)

tot.price <- net.price \* ifelse(public, 1.06, 1.12)

round(price)

}

A continuación, crear una pequeña data frame para probar la función. Por ejemplo:

> clients <- data.frame(

+ hours = c(25, 110, 125, 40),

+ public = c(TRUE,TRUE,FALSE,FALSE)

+)

Ahora, puede utilizar este data frame como argumento para la función priceCalculator(), así:

> with(clients, priceCalculator(hours, public = public))

[1] 1060 4198 5040 1792

# Cómo encadenar sentencias If…Else en R

En algunos casos, usted tendrá que tomar múltiples decisiones en R. Las sentencias if y if...else le dejan con exactamente dos opciones, pero la vida es rara vez tan simple como eso.

Imagine que tiene algunos clientes en el extranjero. Vamos a suponer que cualquier cliente en el extranjero no tiene que pagar el IVA en aras del ejemplo. Esto le deja ahora con tres tipos diferentes de IVA: 12 por ciento para clientes privados, un 6 por ciento para los clientes del sector público, y ninguno para los clientes extranjeros.

La forma más intuitiva para resolver este problema es justamente encadenar las opciones. Si un cliente está viviendo en el extranjero, no se cobra el IVA. De lo contrario, compruebe si el cliente es público o privado, y aplicar el tipo de IVA correspondiente.

Si se define un argumento client  para la función que puede tomar los valores 'abroad', 'public', y'private', puede codificar el algoritmo anterior así:

if(client=='private'){

tot.price <- net.price \* 1.12 # 12% VAT

} else {

if(client=='public'){

tot.price <- net.price \* 1.06 # 6% VAT

} else {

tot.price <- net.price \* 1 # 0% VAT

}

}

Con este código, usted anida la segunda sentencia if...else en la primera sentencia if...else. Eso es perfectamente aceptable y va a trabajar, pero imagínense lo que habría que hacer si usted tiene cuatro o incluso más posibilidades. Anidar una sentencia en una sentencia, y esta a su vez, en una sentencia crea rápidamente un lío enorme.

Por suerte, R permite escribir todo ese código un poco más claramente. Puede encadenar las sentencias if...else de la siguiente manera:

if(client=='private'){

tot.price <- net.price \* 1.12

} else if(client=='public'){

tot.price <- net.price \* 1.06

} else {

tot.price <- net.price

}

En este ejemplo, el encademiento hace una diferencia de sólo dos llaves, pero cuando se tiene más posibilidades, el código se hace más legible. Tenga en cuenta que usted no tiene probar si el argumento client es igual a 'abroad'  (aunque no estaría mal hacer eso). Usted acaba de asumir que si client no tiene cualquiera de los otros dos valores, tiene que ser 'abroad'.

Encadenado sentencias if...else trabajan en un único valor a la vez. Usted no puede utilizar las sentencias if...else encadenadas de una forma vectorizada. Por eso, puede anidar declaraciones múltiples if...else, así:

VAT <- ifelse(client=='private', 1.12,

ifelse(client == 'public', 1.06, 1)

)

tot.price <- net.price \* VAT

Este fragmento de código puede llegar a ser bastante confuso si usted tiene más de tres opciones, sin embargo. La solución a esto es un switch.

# Cómo utilizar bucles con índices en R

La utilización de bucles en R es muy útil, pero usted puede escribir código más eficiente si se recorre no sobre los valores, sino en los índices. Para ello, se reemplaza la sección central en la función con el siguiente código:

nclient <- length(client)

VAT <- numeric(nclient)

for(i in seq\_along(client)){

VAT[i] <- switch(client[i], private=1.12, public=1.06, 1))

}

Aquí hay algunas diferencias con el uso de valores del vector de bucle:

* Se asigna la longitud del vector para el client a la variable nclient.
* Luego se hace un vector numérico IVA que es exactamente del mismo largo que el vector client. Esto es llamado *pre-asignación* de un vector.
* A continuación, se recorre sobre los índices de cliente en lugar del propio vector mediante el uso de la función seq\_along(). En la primera pasada a través del bucle, el primer valor de IVA se establece para ser el resultado de switch() aplicada al primer valor de cliente. En el segundo paso, el segundo valor de IVA es el resultado de switch() aplicada al segundo valor de cliente y así sucesivamente.

Usted puede tener la tentación de sustituir seq\_along(cliente) con el vector 1: nclient, pero eso sería una mala idea. Si el vector client  tiene una longitud de 0, seq\_along (cliente) crea un vector vacío y el código en el bucle nunca se ejecuta. Si utiliza 1: nclient, R crea un vector c(1,0) y un bucle sobre esos dos valores, que le da un resultado completamente equivocado.

Cada vez que usted prolonga un objeto en R, R tiene que copiar todo el objeto y moverlo a un nuevo lugar en la memoria. Esto tiene dos efectos:

* En primer lugar, se ralentiza el código, ya que toda copia lleva tiempo.
* En segundo lugar, como R se mueve continuamente las cosas en la memoria, esta memoria se divide en una gran cantidad de pequeños espacios.

Esto se llama *fragmentación*, y hace que la comunicación entre R y la memoria sea menos suave. Se puede evitar esta fragmentación por *pre-asignación* de memoria como en el ejemplo anterior.

# Cómo recorrer valores en R

A veces, a la hora de tomar decisiones con R, se puede utilizar un único valor para basar su elección sobre él. Se podría aplicar ese código en cada valor que tiene a mano, pero tiene mucho más sentido automatizar esta tarea.

## Construir un bucle for

Como en muchos otros lenguajes de programación, se repite una acción por cada valor en un vector mediante el uso de un bucle for. Construya un bucle for en R como sigue:

for(i in values){

... do something ...

}

Este bucle for se compone de las siguientes partes:

* La palabra clave for, seguido de paréntesis.
* Un identificador entre paréntesis. En este ejemplo, se utiliza i, pero puede ser cualquier nombre de objeto que desee.
* La palabra clave in, que sigue al identificador.
* Un vector con los valores para recorrer. En este código de ejemplo, se utilizan los objetos values pero, una vez más, puede ser cualquier vector que se tiene disponible.
* Un bloque de código entre llaves que tiene que ser llevado a cabo para cada valor en los objetos values.

En el bloque de código, se puede utilizar el identificador. Cada vez que R hace un bucle a través del código, R asigna el valor siguiente en el vector con valores al identificador.

## Calcular los valores en un bucle for

Vamos a echar otro vistazo a la función priceCalculator(). Anteriormente, mostramos algunas posibilidades para adaptar esta función para que pueda aplicar un tipo de IVA diferente para clientes públicos, privados y extranjeros. No se puede utilizar cualquiera de estas opciones de forma vectorizada, pero se puede utilizar un bucle for para que la función calcule el precio de varios clientes a la vez.

### Usar los valores del vector

Adaptar la función priceCalculator() de la siguiente manera:

priceCalculator <- function(hours, pph=40, client){

net.price <- hours \* pph \*

ifelse(hours > 100, 0.9, 1)

VAT <- numeric(0)

for(i in client){

VAT <- c(VAT,switch(i, private=1.12, public=1.06, 1))

}

tot.price <- net.price \* VAT

round(tot.price)

}

La primera y la última parte de la función no han cambiado, pero en el medio, haga lo siguiente:

1. **Crear un vector numérico con longitud 0 y llamarlo IVA.**
2. **Por cada valor en el vector client, aplique switch()  para seleccionar la cantidad correcta de IVA a pagar.**
3. **En cada ronda a través del bucle, añadir el resultado switch()  en el extremo del vector IVA.**

El resultado es un vector IVA que contiene, para cada cliente, el IVA correcto que se debe aplicar. Puede probar esto añadiendo, por ejemplo, una variable type  para el data frame de clientes que creó en la sección anterior de esta manera:

> clients$type <- c('public','abroad','private','abroad')

> priceCalculator(clients$hours, client=clients$type)

[1] 1060 3960 5040 1600

# Cómo hacer más con Bucles en R

R contiene algunos de los mecanismos utilizados en otros lenguajes de programación para manipular bucles:

* La palabra clave next, para pasar a la siguiente iteración de un bucle sin ejecutar el código restante en el bloque de código
* La palabra clave break, para salir de un bucle en cualquier punto dado
* La palabra clave while, para la construcción de un bucle que ejecuta siempre que cierta condición es TRUE.

Puede encontrar más información sobre el uso de estas palabras clave en la página de ayuda ?Control.

Aunque técnicamente se puede utilizar las tres opciones, estas no son usadas ​​con frecuencia. Muchos programadores consideran que el uso de break y next es una mala codificación en cualquier idioma.

Para while, la situación es un poco más compleja. Un bucle while es útil sólo en casos muy específicos, como cuando se generan datos artificiales que tiene que cumplir con ciertas condiciones, o cuando usted escribe sus propios algoritmos de optimización. Pero en muchos casos la funciones integradas de optimización como optim(), optimize(), y nlm() trabajan más rápido que un bucle while – y a menudo más estable. Estas funciones requieren un poco de estudio antes de poder aplicarlas, pero el estudio de las páginas de ayuda ?optim, ?optimize y ?nlm, así como las páginas relacionadas, realmente dan sus frutos.

# Cómo vectorizar sus funciones en R

Las funciones vectorizadas son una característica muy útil de R, pero los programadores que están acostumbrados a otros lenguajes a menudo tienen problemas con este concepto al principio. Una función *vectorizada* no funciona con un solo valor, sino en un vector conjunto de valores al mismo tiempo.

Su reflejo natural como un programador puede ser la de hacer un bucle sobre todos los valores del vector y aplicar la función, pero esto hace innecesaria la vectorización. Confíe en nosotros: Cuando empiece a usar vectorización en R, le va a ayudar a simplificar el código.

Para probar las funciones vectorizadas, usted tiene que hacer un vector. Esto se hace mediante el uso de la función c(), que significa concatenar. Los valores efectivos están separados por comas.

He aquí un ejemplo: Supongamos que Granny juega al baloncesto con su amiga Geraldine y se guarda como calificación el número de canastas de Granny en cada juego. Después de seis juegos, se desea saber cuántas canastas ha hecho Granny en lo que va de la temporada. Usted puede poner estos números en un vector, así:

> baskets.of.Granny <- c(12,4,4,6,9,3)

> baskets.of.Granny

[1] 12 4 4 6 9 3

Para encontrar el número total de canastas hechas por Granny, se escribe lo siguiente:

> sum(baskets.of.Granny)

[1] 38

Usted podría conseguir el mismo resultado al ir al vector por un número y añadir cada nuevo número a la suma de los anteriores, pero ese método podría obligarlo a escribir más código y se necesitaría más tiempo de cálculo. No lo va a notar en tan sólo seis números, pero la diferencia será evidente cuando se tiene que sumar varios miles de ellos.

En este ejemplo de vectorización, una función utiliza el vector completo para darle un resultado. Por supuesto, este ejemplo es trivial (es posible que haya adivinado que sum() lograría el mismo objetivo), pero para otras funciones en R, la vectorización puede ser menos evidente.

Un ejemplo menos obvio de una función vectorizada es la función paste(). Si usted hace una vector con los nombres de los miembros de su familia, paste() puede agregar el apellido a todos ellos con un solo comando, como en el siguiente ejemplo:

> firstnames <- c("Joris", "Carolien", "Koen")

> lastname <- "Meys"

> paste(firstnames,lastname)

[1] "Joris Meys" "Carolien Meys" "Koen Meys"

R toma el vector firstnames  y luego pega el apellido en cada valor. ¿No es genial? En realidad, R combina dos vectores. El segundo vector — en este caso, lastname — es sólo un valor de largo. Ese valor es *reciclado* por la función paste() el tiempo que sea necesario.

También le puede dar a R dos vectores más largos, y R los combinará elemento por elemento, así:

> authors <- c("Andrie","Joris")

> lastnames <- c("de Vries","Meys")

> paste(authors,lastnames)

[1] "Andrie de Vries" "Joris Meys"

No se necesita código complicado. Todo lo que tienes que hacer es construir los vectores y ponerlos en la función.

# Cómo usar la familia Apply de funciones en R

Usar bucles for tiene algunos efectos secundarios muy importantes. Los objetos que se crean en el bucle for después permanecen en el área de trabajo. Los objetos que cambian en el bucle for se cambian en el espacio de trabajo. Si bien estas pueden ser las intenciones de algunos, para otros, se trata de un efecto secundario no deseado de la forma en que los bucles se implementan en R.

Eche un vistazo al siguiente ejemplo:

> songline <- 'Get out of my dreams...'

> for(songline in 1:5) print('...Get into my car!')

Contrariamente a lo que se puede esperar, después de ejecutar este código, el valor de songline  no es la cadena 'Get out of my dreams...', sino el número 5, como se muestra en el resultado a continuación:

> songline

[1] 5

Aunque nunca se cambia explícitamente el valor de songline  en cualquier parte del código, R lo hace implícitamente al realizar el bucle for. En cada iteración, R vuelve a asignar el siguiente valor del vector a songline . . . en el espacio de trabajo!

Al elegir los nombres de las variables y la identificación con prudencia, puede evitar meterse en problemas. Al escribir secuencias de comandos de gran tamaño, que tiene que hacer un poco de contabilidad seria para los nombres con el fin de evitar errores.

Para ser del todo correctos, utilizar un bucle for tiene un efecto sobre el medio ambiente que se trabaja en ese momento. Si usted sólo tiene que utilizar el bucle for de secuencias de comandos que se ejecutan en la consola, los efectos se llevarán a cabo en el área de trabajo. Si utiliza un bucle for en el cuerpo de la función, los efectos se llevarán a cabo dentro del ámbito de esa función.

Aquí están las buenas noticias: R tiene otro sistema de bucle que es muy poderoso, que es al menos tan rápido como el bucle for (y a veces mucho más rápido), y — lo más importante de todo — que no tiene los efectos secundarios de un bucle for. En realidad, este sistema consiste en una familia completa de funciones relacionadas, conocidas como la *familia apply*. Esta familia contiene siete funciones, todas terminan con apply.

## Las características de la familia

Antes de utilizar cualquiera de las funciones de la familia apply, estas son las propiedades más importantes de estas funciones:

* Cada una de las funciones apply lleva al menos dos argumentos: un objeto y otra función. Usted pasa a la función como argumento.
* Ninguna de estas funciones apply tiene efectos secundarios. Esta es la razón principal para usarlas! Si usted puede utilizar cualquier función apply en lugar de un bucle for, utilice la solución aplicable. Tenga en cuenta que los posibles efectos secundarios de la función *aplicada* no son cuidados por la familia apply.
* Toda función apply puede pasar argumentos a la función que se le da como argumento. Lo hace usando el argumento dots.
* Todas las funciones de la familia apply siempre devuelve un resultado. Usar la familia apply sólo tiene sentido si necesita ese resultado. Si desea imprimir mensajes en la consola con print() o cat() por ejemplo, según la familia apply es innecesario.

## Conozca a tres de los miembros

Salude a apply(), sapply(), y lapply(), los miembros más utilizados de la familia apply. Cada una de estas funciones aplica otra función para todos los elementos de un objeto. Qué son esos elementos depende del objeto y la función.

|  |  |  |  |
| --- | --- | --- | --- |
| **Nombre de la función** | **Objetos de trabajo de la función** | **Lo que la function mira como elementos** | **Tipo de resultado** |
| apply | Matriz | Filas o columnas | Vector, matriz, array, o lista |
| Array | Filas, columnas, o cualquier dimensión | Vector, matriz, array, o lista |
| Data frame | Filas o columnas | Vector, matriz, array, o lista |
| sapply | Vector | Elementos | Vector, matriz, o lista |
| Data frame | Variables | Vector, matriz, o lista |
| Lista | Elementos | Vector, matriz, o lista |
| lapply | Vector | Elementos | Lista |
| Data frame | Variables | Lista |
| Lista | Elementos | Lista |

# Cómo aplicar funciones a un vector en R

La función apply() trabaja en cualquier cosa que tenga dimensiones en R, pero ¿qué pasa si usted no tiene dimensiones? Para ello, usted tiene dos funciones relacionadas con la familia apply a su disposición: sapply() y lapply(), lapply significa lista y sapply significa simplificar.

Las dos funciones se utilizan básicamente para lo mismo — la única diferencia es que lapply() siempre devuelve una lista como resultado, mientras que sapply() intenta simplificar el objeto final en lo posible.

Tanto sapply() como lapply() consideran cada valor en el vector para ser un elemento en el cual se puede aplicar una función. Muchas funciones en R trabajan en forma vectorizada, por lo que a menudo no hay necesidad de usar esto.

La función switch(), sin embargo, no funciona en un modo vectorizado. Considere el ejemplo básico siguiente:

> sapply(c('a','b'), switch, a='Hello', b='Goodbye')

a b

"Hello" "Goodbye"

La llamada de sapply() trabaja de forma muy similar a la llamada apply(), aunque usted no tiene un argumento que especifique el índice. He aquí un resumen:

* El primer argumento es el vector en el que están los valores a los que desea aplicar la función — en este caso, el vector c('a', 'b').
* El segundo argumento es el nombre de la función — en este caso, switch.
* Todos los demás argumentos son simplemente los argumentos que se pasan a la función switch .

La función sapply() ahora toma el primer valor 'a' y luego el valor 'b' como el primer argumento para switch(), utilizando los argumentos a='Hello' y b='Goodbye' cada vez como los otros argumentos. Combina ambos resultados en un vector y utiliza los valores de c ('a', 'b') como nombres para el vector resultante.

La función sapply() tiene un argumento USE.NAMES  que puede establecer en FALSE  si no quiere que sapply() use los valores de caracteres como nombres para el resultado.

# Cómo aplicar funciones en filas y columnas en R

En R, puede utilizar la función apply() para aplicar una función sobre cada fila o columna de una matriz o data frame. Esto brinda oportunidades muy útiles.

## Contar en R usando la función apply

Imagine que se cuentan los pájaros en su patio trasero en tres días diferentes y se almacenan los recuentos en una matriz como la siguiente:

> counts <- matrix(c(3,2,4,6,5,1,8,6,1), ncol=3)

> colnames(counts) <- c('sparrow','dove','crow')

> counts

sparrow dove crow

[1,] 3 6 8

[2,] 2 5 6

[3,] 4 1 1

Cada columna representa una especie diferente, y cada fila representa un día diferente. Ahora se quiere averiguar la mayor cantidad por especies en un día determinado. Se podría construir un bucle for para hacerlo, pero usando de apply(),se hace esto en una sola línea de código:

> apply(counts, 2, max)

sparrow dove crow

4 6 8

La función apply() devuelve un vector con el máximo de cada columna y utiliza convenientemente los nombres de columna como nombres para este vector también. Si R no encuentra nombres de la dimensión sobre la que apply() se ejecuta, devuelve un objeto no identificado en su lugar.

Let’s take a look at how this apply() function works. In the previous lines of code, you used three arguments:

Echemos un vistazo a cómo trabaja la función apply(). En las anteriores líneas de código, usted utilizó tres argumentos:

* **El objeto en el que la función tiene que ser aplicado:** En este caso, es la matriz counts.
* **La dimensión o índice en el que la función tiene que ser aplicado:** El número 1 significa modo de fila y el número 2 significa de modo de columna. En este caso, se aplica la función por encima de las columnas. En el caso de más matrices unidimensionales, este índice puede ser mayor que 2.
* **El nombre de la función que ha de aplicarse:** Puede usar comillas alrededor del nombre de la función, pero no tiene que hacerlo. En este caso, se aplica la función max. Tenga en cuenta que no hay paréntesis necesarios después del nombre de la función.

La función apply() divide la matriz en filas. Recuerde que si selecciona una sola fila o columna, R, de forma predeterminada, simplifica eso a un vector. La función apply() utiliza entonces estos vectores, uno por uno como argumento a la función especificada. Así, la función aplicada tiene que ser capaz de tratar con los vectores.

## Añadir argumentos extra a la función apply

Volvamos a nuestro ejemplo de la sección anterior: Imagine que no observó doves el segundo día. Esto significa que, para ese día, no tiene ningún dato, así que hay que ajustar este valor a NA como esto:

> counts[2, 2] <- NA

Si se aplica la función max  en las columnas de esta matriz, se obtiene el siguiente resultado:

> apply(counts,2,max)

sparrow dove crow

4 NA 8

Eso no es lo que se quiere. Con el fin de hacer frente a los valores que faltan, se necesita pasar el argumento na.rm de la función max en la llamada de apply() (véase el Capítulo 4). Por suerte, esto se hace fácilmente en R. Sólo tiene que añadir todos los argumentos extra a la función como argumentos extra a la llamada de apply(), así:

> apply(counts, 2, max, na.rm=TRUE)

sparrow dove crow

4 6 8

Puede pasar cualquier argumento que desee a la función en la llamada de apply(), simplemente hay que agregarlas entre paréntesis después de los tres primeros argumentos.

# Cómo simplificar los resultados (o no) con la función de sapply en R

La función sapply() no siempre devuelve un vector. De hecho, la salida estándar de sapply es una lista, pero esa lista se simplifica ya sea a una matriz o un vector, *si es posible*.

* Si el resultado de la función aplicada a cada elemento de la lista o vector es un solo número, sapply() simplifica el resultado a un vector.
* Si el resultado de la función aplicada a cada elemento de la lista o vector es un vector con exactamente la misma longitud, sapply() simplifica el resultado a una matriz.
* En todos los demás casos, sapply() devuelve una lista (named) con los resultados.

Digamos que usted quiere saber los valores únicos de cada variable en el data frame clients. Para obtener todos los valores únicos de un vector, se utiliza la función unique(). Usted puede obtener el resultado deseado mediante la aplicación de esa función al data frame clients de este modo:

> sapply(clients, unique)

$hours

[1] 25 110 125 40

$public

[1] TRUE FALSE

$type

[1] "public" "abroad" "private"

En las variables hours, usted encontrará cuatro valores únicos; en la variable public, sólo dos; y en la variable type, tres. Debido a que las longitudes de los resultados difieren para cada variable, sapply() no puede simplificar el resultado, por lo que devuelve un nombre de lista.

# Cómo trabajar con archivos y carpetas en R

¿Sabe cómo importar los datos en R y exportar los datos de R? Ahora todo lo que necesita es una idea de donde son almacenados los archivos con R y cómo manipular esos archivos. Cada sesión de R tiene una ubicación predeterminada en la estructura de archivos de su sistema operativo llamado *working directory* (directorio de trabajo).

Es necesario hacer un seguimiento y deliberadamente establecer su directorio de trabajo en cada sesión de R. Si lee o escribe archivos en el disco, estos toman un lugar en el directorio de trabajo. Si no se establece el directorio de trabajo en la ubicación que desea, usted puede fácilmente escribir archivos en una ubicación de archivo no deseado.

La función getwd() le dice cuál es el actual directorio de trabajo:

> getwd()

[1] "F:/git"

Para cambiar el directorio de trabajo, utilice la función setwd(). Asegúrese de introducir el directorio de trabajo como una cadena de caracteres (se encierra entre comillas).

En este ejemplo se muestra cómo cambiar el directorio de trabajo a una carpeta llamada F:/git/roxygen2:

> setwd("F:/git/roxygen2")

> getwd()

[1] "F:/git/roxygen2"

Observe que el separador entre carpetas es la barra inclinada (/), como lo es en los sistemas Linux y Mac. Si utiliza el sistema operativo Windows, la barra diagonal le resultará extraña porque usted está familiarizado con la barra invertida (\) de las carpetas de Windows. Cuando se trabaja en Windows, es necesario utilizar la barra diagonal o evitar las barras invertidas con una doble barra invertida (\\). Compruebe el siguiente código:

> setwd("F:\\git\\stringr")

> getwd()

[1] "F:/git/stringr"

R siempre imprimirá los resultados usando /, pero usted es libre de usar / o \\ a su gusto.

Para evitar tener que lidiar con huir de las barras invertidas en las rutas de archivos, puede utilizar la función file.path() para construir las rutas de archivos que son correctas, independientemente del sistema operativo en que trabaja. Esta función es un poco similar a paste  en el sentido de que se anexará cadenas de caracteres, excepto que el separador es siempre correcto, independientemente de la configuración de su sistema operativo:

> file.path("f:", "git", "surveyor")

[1] "f:/git/surveyor"

A menudo es conveniente utilizar file.path() para establecer el directorio de trabajo. Esto le permite especificar una cascada de letras de unidad y nombres de carpetas, y file.path() entonces ensambla estos en una ruta de archivo adecuado, con el carácter separador correcto:

> setwd(file.path("F:", "git", "roxygen2"))

> getwd()

[1] "F:/git/roxygen2"

También puede utilizar file.path() para especificar rutas de archivos que incluyen el nombre del archivo al final. Sólo tiene que añadir el nombre de archivo para el argumento de la ruta. Por ejemplo, esta es la ruta de acceso al archivo README.md en el paquete roxygen2 instalado en una carpeta local:

> file.path("F:", "git", "roxygen2", "roxygen2", "README.md" )

[1] "F:/git/roxygen2/roxygen2/README.md"

# Cómo manipular archivos en R

En ocasiones, es posible que desee escribir un script en R que atravesará una carpeta determinada y realizará acciones en todos los datos de los archivos o en un subconjunto de los archivos de esa carpeta.

Para obtener una lista de los archivos en una carpeta específica, utilice list.files() o dir() o (. Estas dos funciones hacen exactamente lo mismo, pero por razones de compatibilidad con versiones anteriores, la misma función tiene dos nombres:

> list.files(file.path("F:", "git", "roxygen2"))

[1] "roxygen2" "roxygen2.Rcheck"

[3] "roxygen2\_2.0.tar.gz" "roxygen2\_2.1.tar.gz"

|  |  |
| --- | --- |
| **Función** | **Descripción** |
| Iist.files | Lista de archivos en un directorio. |
| list.dirs | Lista de los subdirectorios de un directorio. |
| file.exists | Comprueba si un determinado archivo existe en un lugar. |
| file.create | Crea un archivo. |
| file.remove | Elimina archivos (y directorios en sistemas operativos Unix). |
| tempfile | Devuelve el nombre de un archivo temporal. Si crea un archivo - por ejemplo, con file.create() o write.table() usando este nombre devuelto - R creará un archivo en una carpeta temporal. |
| tempdir | Devuelve la ruta de archivo de una carpeta temporal del sistema de archivos. |

A continuación, usted ejercerá todo su conocimiento acerca de cómo trabajar con los archivos. En el siguiente ejemplo, primero se crea un archivo temporal, a continuación, se guarda una copia del data frame iris a este archivo. Para comprobar que el archivo está en el disco, a continuación, lea el archivo recién creado a una nueva variable e inspeccione esta variable. Por último, se elimina el archivo temporal del disco.

Comience usando la función tempfile() para devolver un nombre a una cadena de caracteres con el nombre de un archivo en una carpeta temporal del sistema:

> my.file <- tempfile()

> my.file

[1] "C:\\Users\\Andrie\\AppData\\Local\\Temp\\ RtmpGYeLTj\\file14d4366b6095"

Observe que el resultado es puramente una cadena de caracteres, no un archivo. Este archivo no existe aún ninguna parte. A continuación, se guarda una copia del data frame iris a my.file utilizando la función write.csv(). Entonces use list.files() para ver si R ha creado el archivo:

> write.csv(iris, file=my.file)

> list.files(tempdir())

[1] "file14d4366b6095"

Como se puede ver, R creó el archivo. Ahora usted puede utilizar read.csv() para importar los datos a una nueva variable llamada file.iris:

> file.iris <- read.csv(my.file)

Use str() para investigar la estructura de file.iris. Como era de esperar file.iris es un data.frame de 150 observaciones y seis variables. ¿Seis variables, dice usted? Sí, seis, aunque el original iris sólo tiene cinco columnas.

Lo que pasó aquí fue que el valor por defecto del argumento row.names de read.csv() es row.names=TRUE. (Puedes confirmar observando de cerca la Ayuda para ?read.csv()) Por lo tanto, R salvó a los nombres de las filas originales de iris a una nueva columna llamada X:

> str(file.iris)

'data.frame': 150 obs. of 6 variables:

$ X : int 1 2 3 4 5 6 7 8 9 10 ...

$ Sepal.Length: num 5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...

$ Sepal.Width : num 3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...

$ Petal.Length: num 1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ...

$ Petal.Width : num 0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ...

$ Species : Factor w/ 3 levels "setosa","versicolor",..: 1 1 1 1 1 1 1 1 1 1 ...

Para salir del sistema de archivos en su orden original, puede utilizar file.remove() para eliminar el archivo temporal:

> file.remove(my.file)

> list.files(tempdir())

character(0)

Como se puede ver, el resultado de list.files() es una cadena de caracteres vacía, ya que el archivo ya no existe en esa carpeta.

# Cómo leer datos desde Excel a R

Si le pregunta a los usuarios de R cuál es la mejor forma de hacerlo de importar datos directamente desde Microsoft Excel, la mayoría de ellos probablemente responderá que su mejor opción es primero exportar de Excel a un archivo CSV y luego usar read.csv() para importar los datos a R.

De hecho, este sigue siendo la sugerencia en el capítulo 8 de la importación de R y la exportación manual, que dice: "El primer consejo es evitar hacerlo si es posible!" La razón es que muchos de los métodos existentes para la importación de datos desde Excel dependen del software de terceros o las bibliotecas que pueden ser difíciles de configurar, no disponibles en todos los sistemas operativos, o tal vez tienen términos de licencia restrictivos.

Sin embargo, desde febrero de 2011 existe una nueva alternativa: el uso del paquete XLConnect, disponible en [**CRAN**](http://cran.r-project.org/web/packages/XLConnect/index.html). Lo que hace diferente a XLConnect es que utiliza una biblioteca de Java para leer y escribir archivos de Excel. Esto tiene dos ventajas:

* **Se ejecuta en todos los sistemas operativos que soportan Java.** XLConnect está escrito en Java y se ejecuta en Windows, Linux y Mac OS.
* **No hay nada más para cargar.** XLConnect no requiere de otras bibliotecas o software. Si tiene Java instalado, debería funcionar.

XLConnect también puede escribir archivos de Excel, incluyendo el formato de celdas cambiantes, tanto en Excel 97-2003 y Excel 2007/10.

Para saber más sobre [**XLConnect**](http://cran.r-project.org/web/packages/XLConnect/vignettes/XLConnect.pdf), se puede leer el documento *The XLConnect Package*.

Ahora, probablemente está ansioso por empezar con un ejemplo. Suponga que desea leer una hoja de cálculo de Excel en el directorio de usuario llamado Elements.xlsx. En primer lugar, instale y cargue el paquete, y luego cree un objeto con el nombre del archivo:

> install.packages("XLConnect")

> library("XLConnect")

> excel.file <- file.path("~/Elements.xlsx")

Ahora ya está listo para leer una hoja de este libro con la función readWorksheetFromFile(). Debe pasar al menos dos argumentos:

* **archivo**: Una cadena de caracteres con una ruta a una dirección válida .xls o .xlsx.
* **hoja**: Bien un entero que indique la posición de la hoja de trabajo (por ejemplo, sheet = 1) o el nombre de la hoja de trabajo (por ejemplo, sheet = "Sheet2")

Las siguientes dos líneas hacen exactamente lo mismo — ambos importan los datos en la primera hoja (llamada Sheet1):

> elements <- readWorksheetFromFile(excel.file, sheet=1)

> elements <- readWorksheetFromFile(excel.file, sheet="Sheet1")

# Cómo utilizar read.csv() para importar datos en R

Una de las maneras más fáciles y más confiable de obtener datos en R es el uso de archivos de texto, en particular, archivos CSV (comma-separated values). El formato de archivo CSV utiliza comas para separar los diferentes elementos en una línea, y cada línea de datos está en su propia línea en el archivo de texto, lo que hace a los archivos CSV ideales para representar datos tabulares.

El beneficio adicional de archivos CSV es que casi todas las aplicaciones de datos admite la exportación de datos a formato CSV. Este es ciertamente el caso para la mayoría de aplicaciones de hoja de cálculo, como Microsoft Excel y OpenOffice Calc.

En los ejemplos siguientes, suponga que tiene un archivo CSV almacenado en una carpeta conveniente en su sistema de archivos. Para convertir una hoja de cálculo de Excel en formato CSV, debe seleccionar Archivo → Guardar como, que le da la opción de guardar el archivo en una variedad de formatos.

Tenga en cuenta que un archivo CSV puede representar sólo una hoja de cálculo simple. Por último, asegúrese de usar la fila superior de la hoja de trabajo (fila 1) para los encabezados de columna.

En R, se utiliza la función read.csv() para importar datos en formato CSV. Esta función tiene una serie de argumentos, pero el único argumento esencial es file, que especifica la ubicación y el nombre de archivo. Para leer un archivo llamado elements.csv ubicado en F: use read.csv() con file.path:

> elements <- read.csv(file.path("f:", "elements.csv"))

> str(elements)

'data.frame': 10 obs. of 9 variables:

$ Atomic.number: int 1 2 3 4 5 6 7 8 9 10

$ Name : Factor w/ 10 levels "Beryllium","Boron",..: 6 5 7 1 2 3 9 10 4 8

$ Symbol : Factor w/ 10 levels "B","Be","C","F",..: 5 6 7 2 1 3 8 10 4 9

$ Group : int 1 18 1 2 13 14 15 16 17 18

$ Period : int 1 1 2 2 2 2 2 2 2 2

$ Block : Factor w/ 2 levels "p","s": 2 2 2 2 1 1 1 1 1 1

$ State.at.STP : Factor w/ 2 levels "Gas","Solid": 1 1 2 2 2 2 1 1 1 1

$ Occurrence : Factor w/ 1 level "Primordial": 1 1 1 1 1 1 1 1 1 1

$ Description : Factor w/ 6 levels "Alkali metal",..: 6 5 1 2 4 6 6 6 3 5

R importa los datos en un data frame. Como se puede ver, este ejemplo tiene diez observaciones de nueve variables.

Observe que la opción por defecto es la de convertir cadenas de caracteres en factores. Por lo tanto, el nombre de las columnas, Name,Block, State.At.STP, Occurrence, y Description, todas se han convertido en factores. Además, observe que R convierte los espacios en los nombres de columna a puntos (por ejemplo, en la columna State.At.STP).

Esta opción por defecto de convertir cadenas a factores cuando utiliza read.table() puede ser una fuente de gran confusión. Usted es a menudo mejor importador de datos que contienen cadenas de tal manera que las cadenas no se convierten en factores, pero siguen siendo vectores de caracteres. Para importar datos que contienen cadenas, utilice los argumentos stringsAsFactors = false para read.csv() o  read.table():

> elements <- read.csv(file.path("f:", "elements.csv"), stringsAsFactors=FALSE)

> str(elements)

'data.frame': 10 obs. of 9 variables:

$ Atomic.number: int 1 2 3 4 5 6 7 8 9 10

$ Name : chr "Hydrogen" "Helium" "Lithium" "Beryllium" ...

$ Symbol : chr "H" "He" "Li" "Be" ...

$ Group : int 1 18 1 2 13 14 15 16 17 18

$ Period : int 1 1 2 2 2 2 2 2 2 2

$ Block : chr "s" "s" "s" "s" ...

$ State.at.STP : chr "Gas" "Gas" "Solid" "Solid" ...

$ Occurrence : chr "Primordial" "Primordial" "Primordial" "Primordial" ...

$ Description : chr "Non-metal" "Noble gas" "Alkali metal" "Alkaline earth metal" ...

Si usted tiene un archivo con formato de la UE (Unión Europea) (donde las comas se utilizan como separadores decimales y punto y coma se utiliza como separador de campo), tiene que importar a R utilizando la función read.csv2().