

Ciencia de Datos

con

Importa Ordena Transforma Visualiza Comunica



Rubén Sánchez Sancho

Tabla de contenido

Portada	1.1
Prefacio	1.2

Parte 1. Lenguaje R

Sintaxis R	2.1
Expresiones y Asignación	2.1.1
Objetos	2.1.2
Datos Básicos en R	2.1.3
Logical	2.1.3.1
Numeric	2.1.3.2
Integer	2.1.3.3
Character	2.1.3.4
Coerción	2.1.4
Valores Especiales	2.1.5
Inf y -Inf	2.1.5.1
NaN	2.1.5.2
Na	2.1.5.3
Guía Estilo R	2.1.6
Estructuras de Datos	2.2
Vectores	2.2.1
Matrices	2.2.2
Arrays	2.2.3
Listas	2.2.4
Data frames	2.2.5
Estructuras de Control	2.3
Estructuras Condicionales	2.3.1
Estructuras Iterativas	2.3.2
Funciones	2.4
Paquetes	2.5

Parte 2. Manipulación de Datos

Importar Datos en R	3.1
Lectura de Archivos de Texto y Excel	3.1.1
Lectura de Archivos de Programas Estadísticos	3.1.2
Lectura de Archivos Jerarquizados	3.1.3
Lectura de Bases de Datos	3.1.4

Parte 3. Visualización de Datos

Apéndice

Ciencia de Datos con

Importa Ordena Transforma Visualiza Comunica



Rubén Sánchez Sancho

¿Qué hace un "Científico de Datos" y por qué es una profesión tan Sexy?

En 2012, **Davenport y Patil** escribían un [influyente artículo](#) en la Harvard Business Review en la que exponían que el científico de datos era la profesión más sexy del Siglo XXI. Un profesional que combinando conocimientos de matemáticas, estadística y programación, se encarga de analizar los grandes volúmenes de datos. A diferencia de la estadística tradicional que utilizaba muestras, el científico de datos aplica sus conocimientos estadísticos para resolver problemas de negocio aplicando las nuevas tecnologías, que permiten realizar cálculos que hasta ahora no se podían realizar.

Su visión va más allá del **Business Intelligence**, ya que se adentra en el campo del **Big Data**. Las tecnologías de Big Data empiezan a posibilitar que las empresas las adopten y empiecen a poner en valor el análisis de datos en su día a día. Pero, ahí, es cuando se dan cuenta que necesitan algo más que tecnología. La estadística para la construcción de modelos analíticos, las matemáticas para la formulación de los problemas y su expresión codificada para las máquinas, y, el conocimiento de dominio (saber del área funcional de la empresa que lo quiere adoptar, el sector de actividad económica, etc. etc.), se tornan igualmente fundamentales.

Pero, si esto es tan sexy ¿qué hace el científico de datos? Y sobre todo, ¿qué tiene que ver esto con el Big Data y el Business Intelligence? Para responder a ello, a continuación mostramos la representación en formato de diagrama de Venn que hizo [Drew Conway](#) en 2010:



Figura1 - Diagrama de Venn "Ciencia de Datos"

Como podemos apreciar, se trata de una agregación de tres disciplinas que se deben entender bien en este nuevo paradigma que ha traído el Big Data:

-Habilidades informáticas: Partiendo del hecho de que la mayor parte de los datos con los que deberá trabajar el Científico de Datos provendrán de fuentes de datos heterogéneas, por lo tanto, deberá tener las habilidades necesarias para poder extraer, ordenar, analizar y manipular estos datos utilizando distintos lenguajes de programación que le permitan crear los algoritmos necesarios en cada caso concreto.

-Estadística y matemáticas: Una vez extraídos los datos, el Científico de Datos deberá tener los conocimientos matemáticos necesarios para poder interpretarlos y procesarlos mediante las herramientas más adecuadas.

-Conocimiento del dominio: para poder explorar y analizar datos de multitud de orígenes diversos, a menudo inmensos y con estructuras heterogéneas, es necesario conocer el contexto. Los problemas se deben plantear acorde a estas características, para ser capaz de extraer y transmitir recomendaciones en la estrategia empresarial. Podemos sostener, que la Ciencia de los Datos es más una cuestión de plantear bien los problemas que otra cosa, por lo que saber hacer las preguntas correctas con las personas que conocen bien el dominio de aplicación es fundamental.

Como se ilustra en el diagrama de Venn, el Científico de Datos debe ser competente en las 3 áreas básicas descritas anteriormente. En los subconjuntos en los cuales sólo se tienen 2 habilidades de estas 3 áreas, no estaremos hablando de Ciencia de Datos:

-Aprendizaje Supervisado: Sin tener conocimiento del entorno de trabajo, es probable que no se acaben encontrando resultados útiles o adecuados para el proyecto. El objetivo del Científico de Datos no es demostrar su dominio de las herramientas de informáticas o habilidades en Estadística y matemáticas, sino en aplicar estos conocimientos para generar valor y beneficio a su entorno de trabajo. Es decir, un algoritmo de Machine Learning, por muy sofisticado y complejo que sea, no tiene ningún interés en sí mismo, lo importante es el resultado que se obtenga al utilizarlo con una finalidad concreta.

-Investigación Tradicional: La diferencia entre el científico tradicional y el Científico de Datos, radica en gran parte en las habilidades informáticas y conocimientos en lenguajes de programación que debe tener éste último, que le permitirán poder manejar mucha más información y procesarla más rápidamente.

-Zona Comprometida: Un pseudo-Científico de Datos que no tenga destreza en los campos de la estadísticas y las matemáticas, aunque tenga conocimiento del entorno y habilidades informáticas, es probable que procese los datos incorrectamente o los interprete de forma inadecuada, por lo que los resultados de la investigación no tendrán ninguna validez, lo que implicará obtener unas conclusiones erróneas, que incluso podrían perjudicar a futuros proyectos que se basaran en estos resultados.

Estas tres cuestiones (informática y computación, métodos estadísticos y áreas de aplicación/dominio), también fueron citadas por William S. Cleveland en 2001 en su artículo [“Data Science: An Action Plan for Expanding the Technical Areas of the Field of Statistics”](#). Por lo tanto, no es una concepción nueva.

¿Que Aprenderás con este Libro?

El objetivo de este libro es proporcionarte unos fundamentos sólidos en la gran mayoría de herramientas. Nuestro modelo de herramientas necesarias en un proyecto típico de Ciencia de Datos es el que se muestra en la siguiente figura:

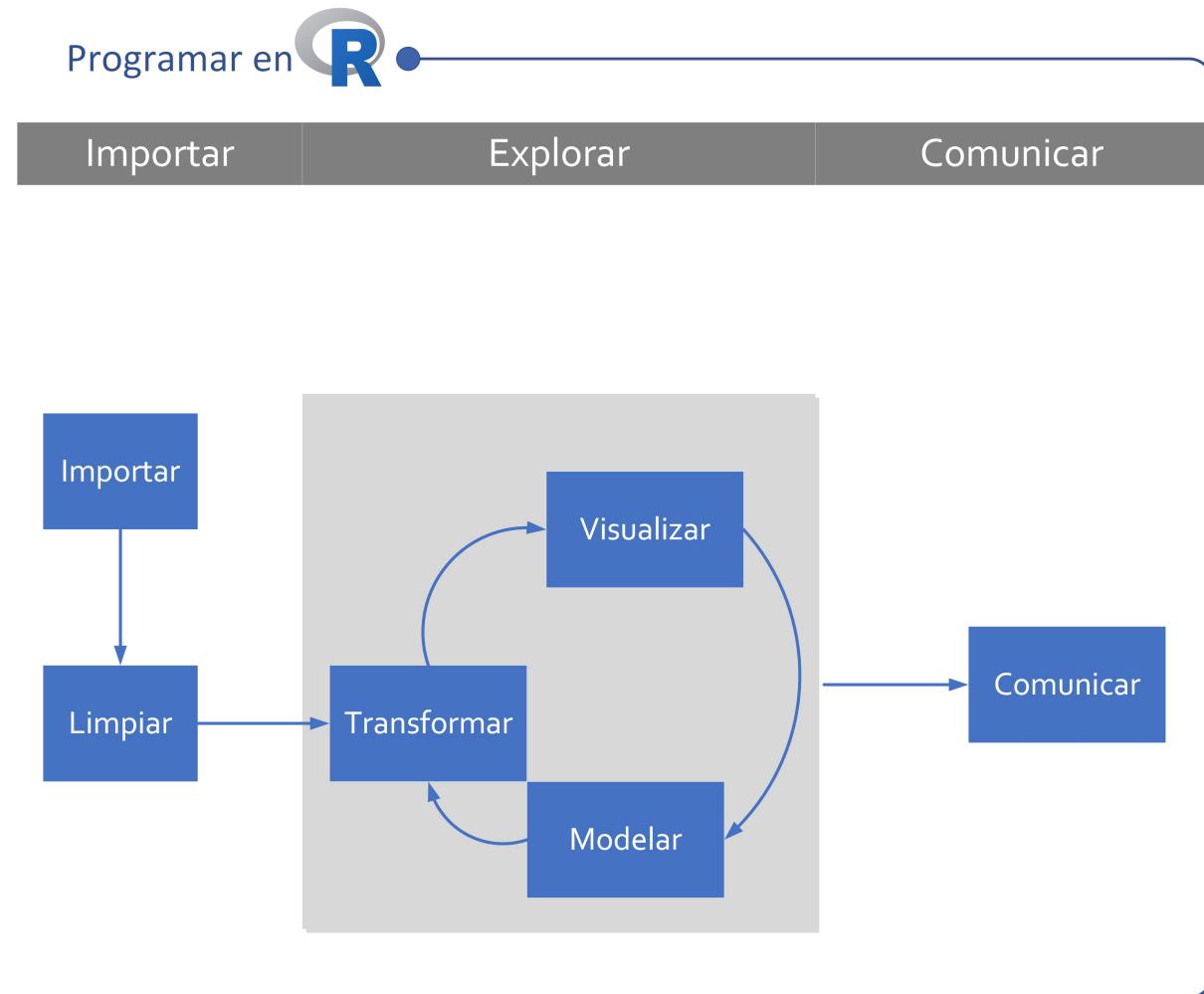


Figura2 - Fases Proyecto en Ciencia de Datos

En primer lugar, será *importar* nuestros datos a R. Con esto queremos decir, que nos encontraremos datos almacenados en archivos, bases de datos, o en una API web, y el objetivo de esta tarea será cargar los datos en un dataframe.

Una vez hemos importado nuestros datos, la siguiente tarea será "*tidy*" nuestros datos. El objetivo de esta tarea es almacenar nuestros datos en un formato consistente en el que coincida la semántica del conjunto de datos con el medio en que están almacenados. En resumen, nuestros datos estarán en formato tidy cuando, cada variable se encuentre en una columna y cada observación en su propia fila.

A continuación, una tarea común es *transformar* nuestros datos. Transformar nuestros datos incluye filtrar las observaciones de nuestro interés (como por ejemplo, todo la gente en una ciudad, o todos los datos del último año), creación de nuevas variables resultado del cálculo de funciones de variables existentes (como por ejemplo, calcular la velocidad con la velocidad y tiempo) y, calcular un conjunto de indicadores estadísticos en un resumen (como medias o desviaciones típicas).

Después, visualizaremos y modelaremos nuestros datos. En este libro únicamente trataremos la *visualización*.

La visualización es fundamental en la actividad humana. Un visualización bien hecha nos muestra cosas que no esperábamos, o nos conduce a formularnos nuevas cuestiones en nuestros datos. Además, puede indicarnos que no estamos formulando las cuestiones correctas, o que necesitamos recoger nuevos datos.

El último paso en la ciencia de datos es la *comunicación*, una parte absolutamente crítica en un proyecto de análisis de datos. Indiscutiblemente, no importa lo bien que hayamos modelado o visualizado nuestros datos, si posteriormente no somos capaces de comunicar nuestros resultados con los demás.

Instalación de las Herramientas

Para seguir este curso haremos uso de tres herramientas: R, RStudio y la colección de paquetes *tidyverse*.

Instalación de R

Para descargar R, lo haremos desde CRAN, un conjunto de servidores espejo distribuidos a lo largo del mundo y usado para distribuir R y paquetes R. La forma mas fácil de instalar R es desde <https://cloud.r-project.org/>.

Instalación de RStudio

RStudio es un entorno integrado de desarrollo, o IDE, para facilitarnos la tarea de programación. Podemos descargo e instalarlo desde <http://www.rstudio.com/download>.

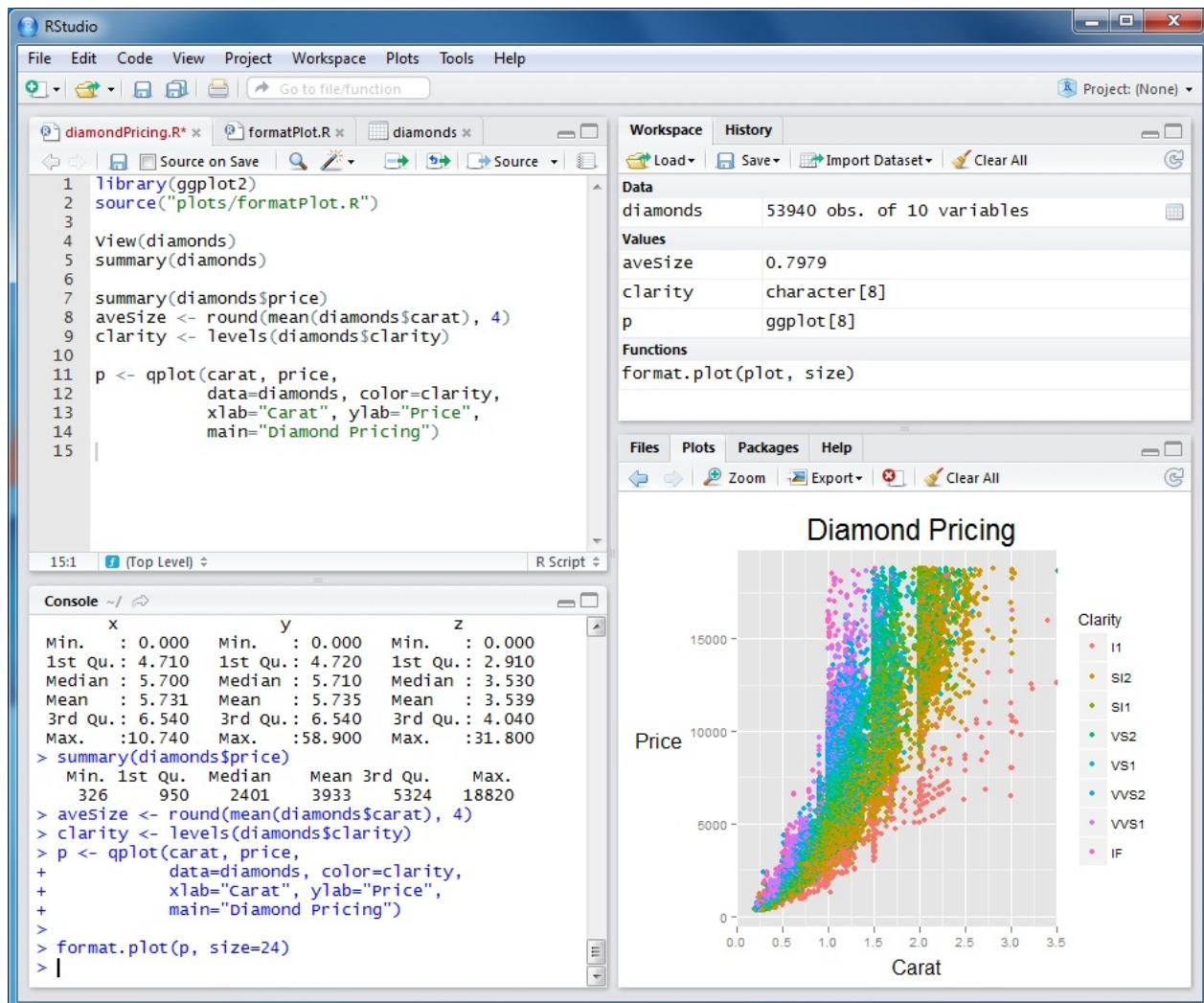


Figura3 - RStudio el IDE para R

Instalacion del Ecosistema Tidyverse

Además, también necesitamos instalar algunos paquetes R. Un *paquete* R es una colección de funciones, datos y documentación que amplían las capacidades base de R. El uso de paquetes es una pieza clave para usar R satisfactoriamente. La mayoría de paquetes que aprenderemos en este texto son parte del ecosistema tidyverse. Los paquetes en tidyverse comparten la misma filosofía en el formato de datos y programación, y están diseñados para trabajar de forma conjunta cubriendo todas las tareas en el análisis de un proyecto típico en ciencia de datos.

Flujo de Trabajo con

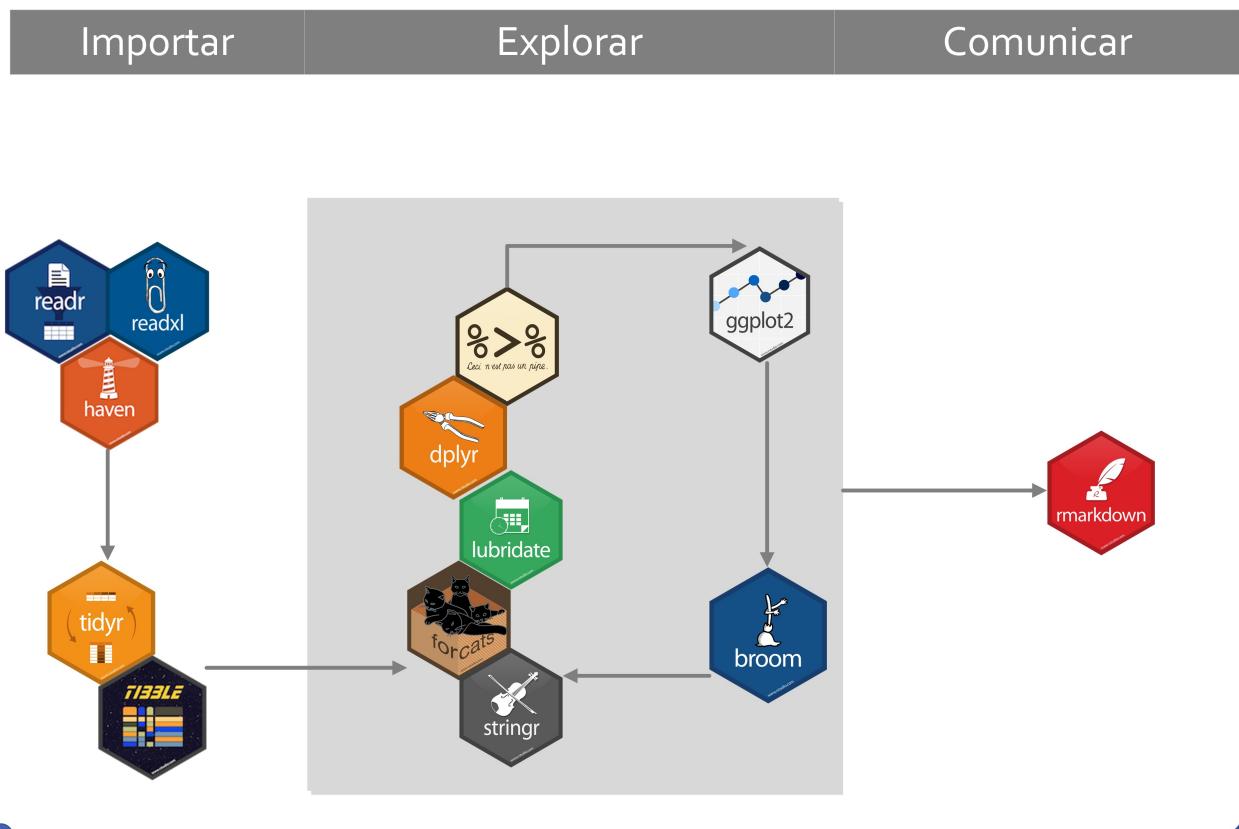


Figura4 - Tidyverse para Proyecto Típico Ciencia de Datos

Podemos instalar el ecosistema tidyverse al completo con tan sólo una única línea de código:

```
install.packages("tidyverse")
```

Para poder hacer uso de las funciones, objetos y archivos de ayuda del paquete necesitaremos además, cargarlo en la sesión R. Esto lo conseguiremos mediante la ayuda de la función `library()`:

```
library(tidyverse)
## -- Attaching packages ----- tidyverse 1.2.1 --
## v ggplot2 2.2.1     v purrr   0.2.4
## v tibble  1.3.4     v dplyr    0.7.4
## v tidyverse 0.7.2    v stringr  1.2.0
## v readr   1.1.1     vforcats 0.2.0
## -- Conflicts ----- tidyverse_conflicts() --
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()   masks stats::lag()
```

Sintaxis R

Aprender un lenguaje de programación es como aprender un segundo idioma (sólo que más simple). Si visitamos un país extranjero, podríamos aprender frases para salir del paso sin entender cómo está estructurado el lenguaje. De forma similar, si sólo queremos hacer un par de cosas con R (p. ej. dibujar gráficos), probablemente sea suficiente con estudiar a partir de unos cuantos ejemplos.

Sin embargo, si lo que queremos es aprender realmente bien un nuevo lenguaje, tendremos que aprender su sintaxis y gramática: conjugación verbos, estructuras gramaticales, etc. Lo mismo es cierto para R: si queremos aprender a programar en R, tendremos que aprender la sintaxis y gramática.

Objetivos

Después de leer este capítulo, deberíamos:

- Conocer las diferentes construcciones que ofrece R para escribir expresiones.
- Escribir expresiones válidas.
- Comprender qué son los objetos.
- Ser capaz de examinar el tipo y clase de un objeto.
- Conocer los diferentes tipos de datos básicos en R.
- Entender el concepto de coerción.
- Comprender la importancia de adoptar un buen estilo para nuestro código.

Expresiones y Asignación

El código R está compuesto por una serie de *expresiones*. Ejemplo de expresiones en R incluyen, expresiones aritméticas, [instrucciones de control](#) (condicionales e iterativas) e instrucciones de asignación. A continuación se muestran ejemplos de expresiones:

```
# Expresión aritmética  
127%%10  
## [1] 7
```

```
# Instrucción condicional  
if (1 > 2) "mayor" else "menor"  
## [1] "menor"
```

```
# Instrucción asignación  
x <- 1
```

Instrucciones de Asignación

Podemos crear nuevos objetos con el operador de asignación `<-`. Todas las instrucciones de asignación donde creamos objetos, tienen la misma forma:

```
nombre_objeto <- valor
```

En lenguaje natural podemos traducirlo como a "este objeto asigne el siguiente valor.

Construcciones para Agrupar Expresiones

R proporciona diferentes construcciones para agrupar expresiones:

- punto y coma
- paréntesis
- llaves

Separando Expresiones con punto y coma

Podemos escribir una serie de expresiones en líneas separadas:

```
x <- 1  
y <- 2  
z <- 3
```

Alternativamente, podemos colocarlas en la misma línea, separadas por punto y coma:

```
x <- 1  
y <- 2  
z <- 3
```

Paréntesis

La notación con paréntesis devuelve el resultado de evaluar la expresión dentro del paréntesis:

```
(x <- 1)  
## [1] 1
```

```
# es equivalente  
x <- 1  
x  
## [1] 1
```

Agrupar expresiones con paréntesis puede ser usado para modificar la prioridad en los operadores:

```
# La multiplicación tiene prioridad sobre la suma  
2 * 5 + 1  
## [1] 11
```

```
# En este caso, se calculará primero la suma y después se multiplicará  
2 * (5 + 1)  
## [1] 12
```

Llaves

Las llaves son usadas para evaluar una serie de expresiones (separadas por nuevas líneas o punto y comas) y, nos devolverán el resultado de la última expresión:

```
{  
  x <- 1  
  y <- 2  
  x + y  
}  
## [1] 3
```

Usado para agrupar un conjunto de expresiones en el cuerpo de una función:

```
f <- function() {  
  x <- 1  
  y <- 2  
  x + y  
}  
f()  
## [1] 3
```

Objetos

Todo el código R manipula objetos. El *objeto*, es el concepto principal sobre el cual se fundamenta la tecnología orientada a objetos. Un objeto puede ser visto como una entidad que posee atributos y efectúa acciones. Ejemplos de objetos en R incluyen las funciones, *symbols* (nombre objetos) e incluso las expresiones. A continuación se muestran algunos ejemplos de objetos:

```
# Una función es un objeto
function() {
  x <- 1
  y <- 2
  x + y
}
## function()
##   x <- 1
##   y <- 2
##   x + y
## }
## <environment: 0x00000000241a0b28>
```

```
# también su nombre (symbol)
f <- function(x, y) {
  x + y
}
f(1, 2)
## [1] 3
```

```
# incluso las expresiones
{
  x <- 1
  y <- 2
  x + y
}
## [1] 3
```

Symbols

Formalmente, los nombres de las variables en R se designan como *symbol*. Cuando realizamos una asignación de un objeto al nombre de una variable, estamos en realidad asignando el objeto a un symbol. Por ejemplo, la instrucción:

```
x <- 1
```

asigna el symbol "x" al objeto "1" en el entorno actual.

Cada Objeto es Miembro de una Clase

Todos los objetos en R tienen una clase, la cual define que información contiene y como usarlo. Por ejemplo, la mayoría de números son *numeric* ([lo veremos en la siguiente sección](#)), y los valores lógicos pertenecen a la clase *logical*. En particular, puesto que R no dispone de un tipo para representar escalares, los vectores de números son numeric y los vectores de valores lógicos son logical. El tipo de dato "más pequeño" en R es un vector.

Podemos encontrar la clase de un objeto mediante la función `class(objeto)` :

```
vector_numerico <- c(1, 2, 3, 4, 5)
class(vector_numerico)
## [1] "numeric"
```

Cada Objeto tiene un Tipo

Cada objeto en R tiene un tipo. El tipo define como es almacenado el objeto en R.

Podemos conocer el tipo de objeto con la función `typeof(objeto)` :

```
vector_numerico <- c(1, 2, 3, 4, 5)
typeof(vector_numerico)
## [1] "double"
```

Comprobar la Clase de un Objeto en *scripts*

El uso de la función `class()` es útil para examinar nuestros objetos en un trabajo interactivo con la consola de R, pero si necesitamos comprobar el tipo de un objeto en nuestros *scripts*, es mejor el uso de la función `is()`. En una situación típica, nuestro test se parecerá al siguiente:

```
if (!is(x, "alguna_clase")) {
  # alguna acción correctiva
}
```

La mayoría de las clases tienen su propia función `is.*()`, utilizar esta función es mas efectivo que el uso de la función general `is()`. Por ejemplo:

```
is.character("Ciencia de Datos con R")
## [1] TRUE
```

Podemos ver una lista completa de las funciones `is()` en el paquete `base` mediante la siguiente instrucción:

```
ls(pattern = "^\is", baseenv())
```

Nombres de los Objetos

Los nombres de los objetos pueden contener letras, números, puntos y guiones bajos, pero no pueden empezar con un número o un punto seguido de un número. Además, no podemos hacer uso de las palabras reservadas del lenguaje como `if` y `for`. La instrucción `?make.names` describe en detalle lo que está y no permitido.

En resumen, los nombres de nuestros objetos :

- Pueden contener:
 - letras
 - números
 - guión bajo (-)
 - punto (.)
- Deben empezar con una letra.
- No podemos utilizar las palabras reservadas del lenguaje.

Para la comunidad hispanohablante, cabe mencionar que para que nuestro código sea lo mas portable posible no debemos utilizar acentos y la letra "ñ", es decir, debemos limitarnos a los caracteres de la "a" a la "z" (y de la "A" a la "Z").

Ejemplos:

- ruben
- RUBEN123
- las_variables_pueden_contener_guiones
- las.variables.pueden.contener.puntos
- Ruben_123.0

Convención para Nombres con Varias Palabras

Puesto que queremos que nuestros objetos tengan nombres descriptivos, necesitaremos una convención para objetos con múltiples palabras. Recomendamos usar el método **snake_case** dónde sepáramos las palabras en minúsculas con el guión bajo:

Ejemplos:

- yo_uso_snake_case
- otraGenteUsaCamelCase
- alguna.gente.usa.puntos
- Y_otra.Gente.RENUNICAconvención

Datos Básicos en R

R proporciona cuatro tipos básicos de datos, también conocidos como vectores atómicos.

- logical
- numeric
- integer
- character

Logical

El tipo *logical* es la forma que tiene R para los datos binarios. Usados en test lógicos son conocidos como valores booleanos y toman los valores **TRUE** y **FALSE**. TRUE y FALSE pueden ser abreviados con las T y F en mayúsculas respectivamente. Sin embargo, recomendamos utilizar la versión completa de TRUE y FALSE.

```
3 < 4
## [1] TRUE
```

```
class(TRUE)
## [1] "logical"
```

```
class(T)
## [1] "logical"
```

Es posible construir condiciones lógicas utilizando los operadores `&`, `|` y `!` (en lenguaje formal, *y*, *o* y *no* respectivamente). La comparación de valores se lleva a cabo mediante `==` (es igual a) y `!=` (es distinto de):

```
a <- 2
b <- 4
a == b # ¿es igual a b?
## [1] FALSE
```

```
a != b # ¿es a distindo de b?
## [1] TRUE
```

```
(a < 3) & (b < 5) # ¿es a menor que 3 y b menor que 3?
## [1] TRUE
```

```
(a < 1) | (b < 3) # ¿es a menor que 1 o b menor que 3?
## [1] FALSE
```

A continuación se muestran los operadores de comparación y lógicos en R:

	?Comparision
<	Menor que
>	Mayor que
==	Igual a
<=	Menor que o igual a
>=	Mayor que o igual a
!=	Distinto a
%in%	Pertenece a
is.na()	Es el valor NA?
!is.na()	Valores distintos a NA

Tabla 1: Operadores de comparación

	?base::Logic
&	y (booleano)
	o (booleano)
!	no
any	Cualquiera verdadero
all	Todos verdaderos

Tabla 2: Operadores lógicos

Para mas información sobre la sintaxis de los operadores y su precedencia consultar la documentación R:

```
# Sintaxis de comparación
`?`(Comparison)
# Operadores lógicos
`?`(base::Logic)
```

Numeric

Para representar los números reales R proporciona el tipo *numeric*. Podemos realizar toda clase de operaciones con ellos como por ejemplo sumas, restas, multiplicaciones, divisiones y utilizarlos en el amplio catálogo de funciones matemáticas con las que cuenta R:

```
mi_altura_en_cm <- 180  
mi_altura_en_cm  
## [1] 180
```

```
mi_peso <- 79.5  
mi_peso  
## [1] 79.5
```

```
IMC <- mi_peso/mi_altura_en_cm^2  
IMC  
## [1] 0.002453704
```

```
round(mi_peso)  
## [1] 80
```

Integer

Un tipo especial de numeric es el *integer*. Este es el modo de representar los números enteros en R. Para especificar que un número es entero, debemos añadir la letra L en mayúscula como sufijo.

```
mi_edad <- 40L  
mi_edad  
## [1] 40
```

En el ejemplo anterior, no podemos apreciar la diferencia entre el número real y el número entero. Sin embargo, con la función `class()` podemos comprobar esta diferencia:

```
class(40)  
## [1] "numeric"
```

```
class(40L)
## [1] "integer"
```

En lugar de preguntar por la clase de una variable mediante la función `class()`, podemos utilizar las funciones `is.*()` para comprobar si un objeto es realmente de un cierto tipo. Por ejemplo, para comprobar si una variable es `numeric`, usaremos la función `is.numeric()`:

```
is.numeric(40)
## [1] TRUE
```

```
is.numeric(40L)
## [1] TRUE
```

Para comprobar si una variable es `integer`, usaremos la función `is.integer()`:

```
is.integer(40)
## [1] FALSE
```

```
is.integer(40L)
## [1] TRUE
```

Como podemos ver en el ejemplo anterior los números reales son `numeric`, pero no todos los `numeric` son enteros.

Character

Cualquier dato alfanumérico (o cadenas, *strings* en inglés; todo aquello que no sea un número es una cadena) será interpretado por R como *character*. Por ejemplo:

```
"Ciencia de Datos con R"
## [1] "Ciencia de Datos con R"
```

Otros Tipos de Datos Básicos

Por último, cabe señalar que existen otros tipos de datos básicos en R, como el *double* que es un tipo numérico de doble precisión. El *complex* para números complejos y el *raw* para almacenar bytes. Sin embargo, no los trataremos en este curso puesto que raramente se

utilizan en el análisis de datos

- Double (doble precisión)
- Complex (números complejos)
- Raw (almacenar bytes)

Coerción

La *coerción* es una característica de los lenguajes de programación que permite, implícita o explícitamente, convertir un elemento de un tipo de datos en otro, sin tener en cuenta la comprobación de tipos.

Coerción Implícita

Cuando llamamos a una función con un argumento de un tipo erróneo, R intentará convertir los elementos de un tipo de datos en otro, sin tener en cuenta la comprobación de tipos. Por ejemplo, supongamos que definimos un vector `v` como en el ejemplo:

```
v <- c(1, 2, 3, 4, 5)
v
## [1] 1 2 3 4 5
```

```
typeof(v)
## [1] "double"
```

```
class(v)
## [1] "numeric"
```

Si cambiamos el segundo elemento del vector con la palabra "coercion". R cambiará la clase del objeto a `character` y todos los elementos del vector a `char` como podemos ver en el siguiente ejemplo:

```
v[2] <- "coercion"
```

```
typeof(v)
## [1] "character"
```

```
class(v)
## [1] "character"
```

Cuando un vector lógico es convertido a un integer o double, `TRUE` es cambiado a 1 y `FALSE` a 0:

```
v <- c(FALSE, TRUE, FALSE)
as.numeric(v)
## [1] 0 1 0
```

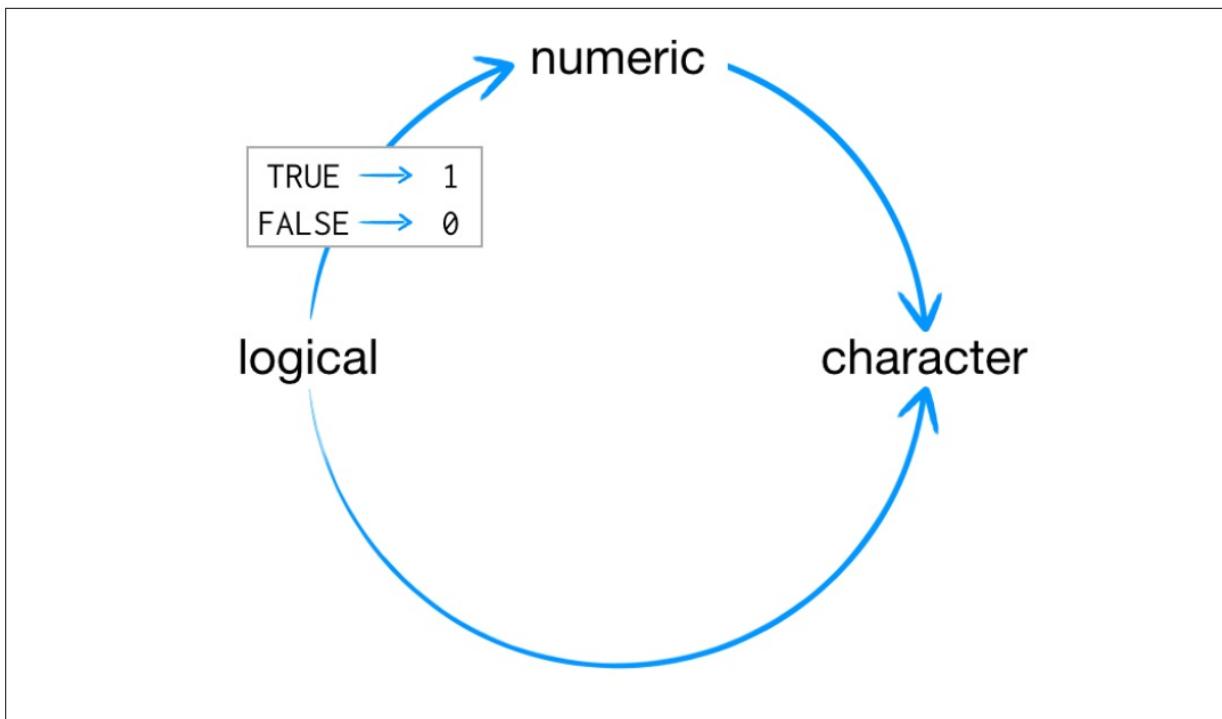


Figura 1 - Coerción

A continuación se muestra un resumen de las reglas para la coerción:

- Los valores lógicos son transformados a números: `TRUE` será cambiado a **1** y `FALSE` a **0**.
- Los valores son convertidos al tipo más simple.
- El orden es: `logical` < `integer` < `numeric` < `complex` < `character` < `list`.
- Los objetos de tipo `raw` no son convertidos a otros tipos.
- Los atributos de los objetos son borrados cuando un objeto es cambiado de un tipo a otro.

Coerción Explícita

Los objetos pueden ser convertidos en otro tipo de forma explícita mediante el uso de la función `as.()`*

Ejemplos

```
v <- c(1, 2, 3, 4, 5)
class(v)
## [1] "numeric"

as.logical(v)
## [1] TRUE TRUE TRUE TRUE TRUE
```

```
as.character(v)
## [1] "1" "2" "3" "4" "5"
```

En ocasiones, la conversión no puede ser llevada a cabo, en este caso R devuelve `NA`.

```
v <- c("a", "b", "c")
as.numeric(v)
## Warning: NAs introduced by coercion
## [1] NA NA NA

as.logical(v)
## [1] NA NA NA
```

En resumen, la mayoría de las funciones producen un error cuando el tipo de datos que esperan no coincide con los que pasamos como argumentos. En esta situación tenemos dos posibilidades:

- Comprobar el tipo de datos utilizando las funciones `is.*()`, que nos responden con un valor lógico.
- Forzar el tipo de datos deseados coercionando, para lo cual podemos utilizar funciones del tipo `as.*()`, que fuerzan el tipo de datos.

A continuación se muestra una lista con los tipos más importantes que se pueden comprobar o forzar:

Tipo	Comprobación	Coerción
array	`is.array()`	`as.array()`
character	`is.character()`	`as.character()`
complex	`is.complex()`	`as.complex()`
double	`is.double()`	`as.double()`
factor	`is.factor()`	`as.factor()`
integer	`is.integer()`	`as.integer()`
list	`is.list()`	`as.list()`
logical	`is.logical()`	`as.logical()`
matrix	`is.matrix()`	`as.matrix()`
NA	`is.na()`	`as.na()`
NaN	`is.nan()`	`as.nan()`
NULL	`is.null()`	`as.null()`
numeric	`is.numeric()`	`as.numeric()`
vector	`is.vector()`	`as.vector()`

Tabla 1: Comprobación y coerción de los tipos más importantes

Podemos ver una lista completa de todas las funciones `is.()` en el paquete `base` mediante:

```
ls(pattern = "^\is", baseenv())
```

Asimismo, para obtener las funciones `as.*()` podemos hacerlo mediante la siguiente instrucción:

```
ls(pattern = "^\as", baseenv())
```

Valores Especiales

Para ayudarnos con los cálculos aritméticos R, soporta cuatro valores numéricos especiales:

- Inf
- -Inf
- NaN
- Na

Los dos primeros, son la forma positiva y negativa para valores infinitos. **NaN** (del inglés, "not-a-number") significa que nuestro cálculo o no tiene sentido matemático o que podría no haberse realizado correctamente. **NA** (del inglés, "not available") representa un valor desconocido.

Inf y -Inf

Si una computación resulta en un número que es demasiado grande, R devolverá `Inf` para un numero positivo y `-Inf` para un número negativo (esto es un valor infinito positivo y infinito negativo, respectivamente):

```
2^1024
## [1] Inf
```

```
-2^1024
## [1] -Inf
```

Esto es también cierto cuando hacemos la división entre 0:

```
1/0
## [1] Inf
```

NaN

En ocasiones, una computación producirá un resultado que no tiene sentido. En estos casos, R devolverá `NaN` (del inglés, "not a number"):

```
Inf - Inf  
## [1] NaN
```

```
0/0  
## [1] NaN
```

NA

En R, los valores `NA` son usados para representar valores desconocidos. (`NA` es la abreviación "not available"). Nos encontraremos valores `NA` en texto importado a R (representando valores desconocidos) o datos importados desde bases de datos (para reemplazar valores `NULL`).

A modo de ejemplo, si el vector `peso` recoge los pesos de 5 personas, habiéndose perdido el cuarto valor, se codificaría como:

```
peso <- c(77, 68, 85, NA, 73)
```

Si pretendemos calcular el peso medio, obtendremos como resultado un valor perdido:

```
mean(peso)  
## [1] NA
```

Si, en cualquier caso, deseamos calcular la media de los pesos efectivamente disponibles, utilizaríamos la opción de *eliminar valores perdidos* (del inglés, `NA remove`) que se declara como `na.rm=TRUE`:

```
mean(peso, na.rm = TRUE)  
## [1] 75.75
```

Guía Estilo R

Los estándares para el estilo en el código no son lo mismo que la sintaxis del lenguaje, aunque están estrechamente relacionados. Es importante adoptar un buen estilo para maximizar la legibilidad de nuestro código para que su mantenimiento sea más fácil tanto para nosotros como otros programadores.

En este curso, hemos adoptado la [Guía Estilo de Hadley Wickham](#) que incluye en su libro *Advanced R* y que está basada en la [Guía Estilo R de Google](#). A continuación se muestra un resumen de sus sugerencias:

Nombres de Archivos

En los nombres de los archivos debemos usar nombres descriptivos, no tienen que incluir caracteres especiales (tildes, eñes) y asegurarse que su extensión es ".R"

```
# Bien
regresion_lineal.R
lectura_datos.R

# Mal
regresión_lineal.R
version dellunes.R
```

Nombres de los Objetos

"Hay únicamente dos problemas realmente duros en informática; el primero es la invalidación de cachés, y el segundo darles nombres apropiados a > lascosas"

David Wheeler

Los nombres de los objetos y funciones deben ser en minúscula y hemos de separar con guión bajo (_) aquellos nombres compuestos de varias palabras. Wickham recomienda usar sustantivos para los nombres de los objetos y verbos para las funciones. Debemos elegir nombres concisos y descriptivos para nuestros objetos (tarea nada fácil). Para la comunidad hispanohablante recomendamos no utilizar tildes y eñes en los nombres de los objetos.

```
# Bien
primer_cuartil
cuartil_1

# Mal
CuartilUno
C1
está_en_el_primer_cuartil
```

Espacios

Antes y después de todos los operadores infijos (=, +, <-, etc) hay que colocar un espacio. No hay que dejar espacios después de una coma, pero una coma siempre es seguida de un espacio.

```
# Bien
hipotenusa <- sqrt( a ^ 2 + b ^ 2)
v <- c(1, 2, 3)
```

```
hipotenusa <- sqrt(a^2 + b^2)
v <- c(1, 2, 3)
```

La excepción a esta regla es cuando usamos el operador de secuencia :

```
# Bien
vector <- 1:10

# Mal
vector <- 1 : 10
```

Antes del paréntesis izquierdo va un espacio, a no ser que estemos llamando a una función.

```
# Bien
if (1 > 2) "mayor" else "menor"
mean(1:10)

# Mal
if(1 > 2) "mayor" else "menor"
mean (1:10)
```

Punto y coma

Se recomienda no usar punto y coma en nuestras expresiones.

```
# Bien
x <- 1
y <- 2
z <- 3

# Mal
x <- 1; y <- 2; z <- 3
```

Llaves

Abrir una llave nunca debería ocurrir en su propia línea y siempre se sigue con una línea nueva. La llave que cierra siempre tiene que ir su propia línea, a menos que se trate de `else`. Siempre usaremos llaves aunque el cuerpo de la estructura de control contenga sólo una instrucción.

Dentro de las llaves utilizaremos sangrado para indicar que se trata de un bloque de instrucciones.

```
# Bien
if (1 < 2) {
  "mayor"
} else {
  "menor"
}

if (x == 0) {
  "cero"
}

# Mal
if (1 < 2)
{
  "mayor"
}
else {
  "menor"
}

if (x == 0)
  "cero"
```

Longitud de Líneas

La longitud de nuestras líneas debe tener como máximo 80 caracteres. Si usamos RStudio podemos activarlo en:

Tools -> Global Options -> Code -> Display

seleccionando la casilla

[] Show margin Margin column [80]

nos aparecerá una guía en el editor indicando el límite hasta donde podemos escribir nuestra línea.

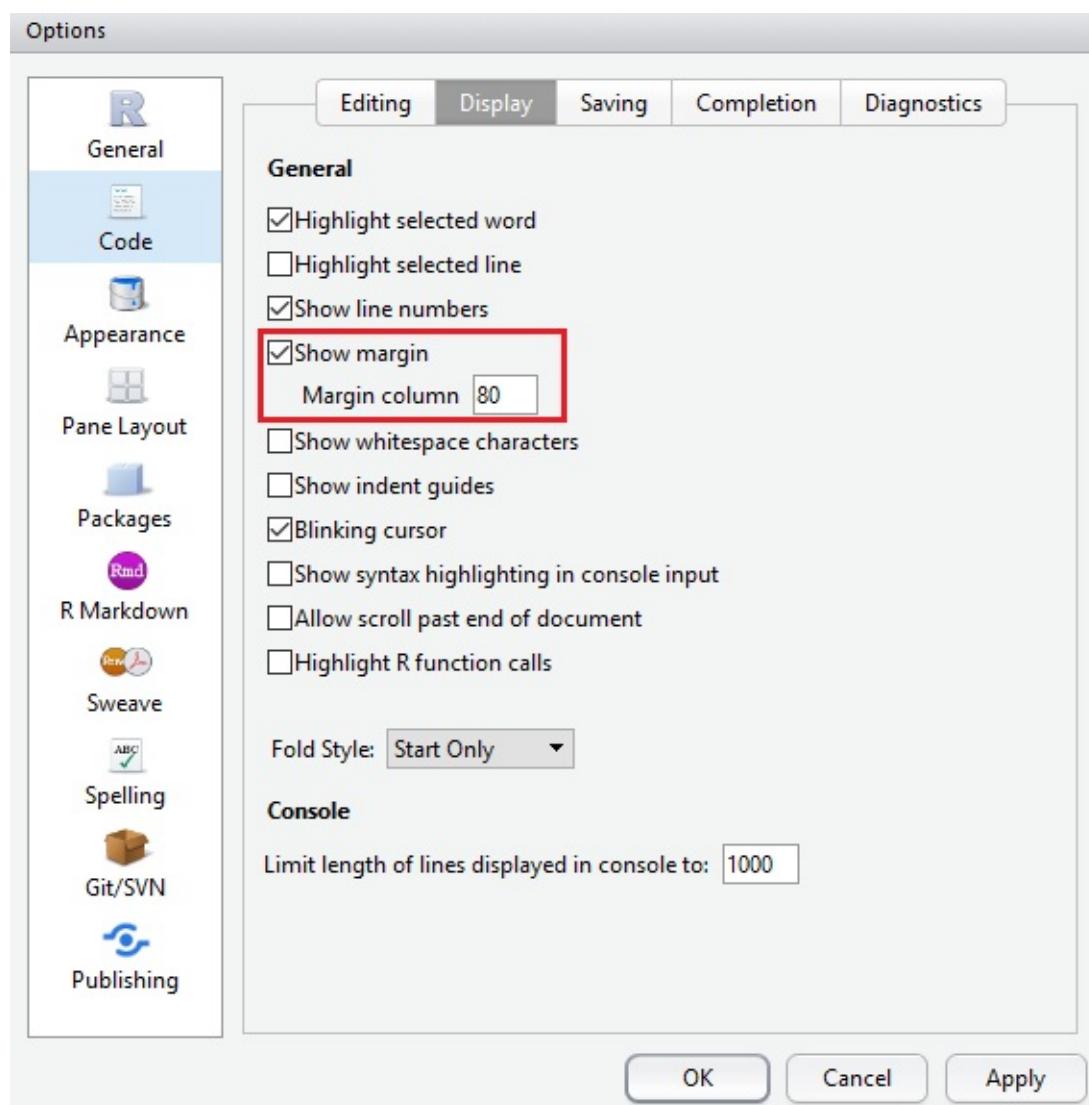


Figura1 - Longitud linea en RStudio

Sangría

Respecto a la sangría en nuestro código se sugiere usar dos espacios para añadir sangría a nuestro código y nunca la tecla `TAB`. En el caso que usemos RStudio podemos configurar la tecla `TAB` para que utilice dos espacios en:

Tools -> Global Options -> Code -> Editing

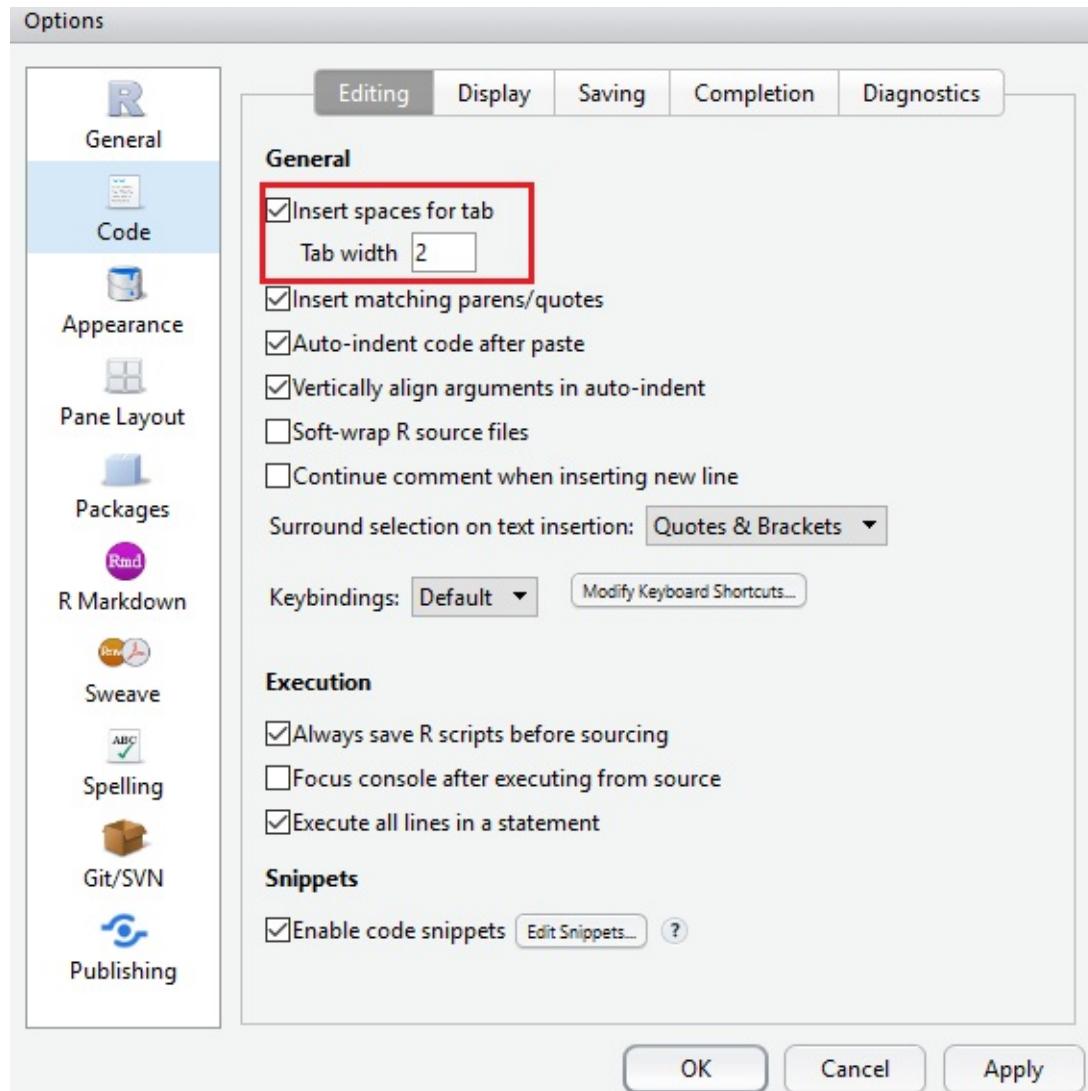


Figura2 - Tabulador en RStudio

Asignación

En la instrucción de asignación se aconseja el uso del operador `<-` en lugar de `=`, reservando este último para declarar argumentos dentro de las funciones.

```
# Bien
x <- 2

# Mal
x = 2
```

Comentarios

Recomendamos documentar el código mediante el uso de comentarios. Cada linea debe comenzar con el signo almohadilla (#) seguido de un espacio en blanco. Los comentarios deben explicar el porqué, y no el qué.

Para separa y estructurar el código en bloques es común el uso de más de una almohadilla # . RStudio ofrece las características de *Code Folding* y *Sections* que nos permite mostrar y ocultar bloques de código permitiendo una navegación más fácil en nuestros archivos.

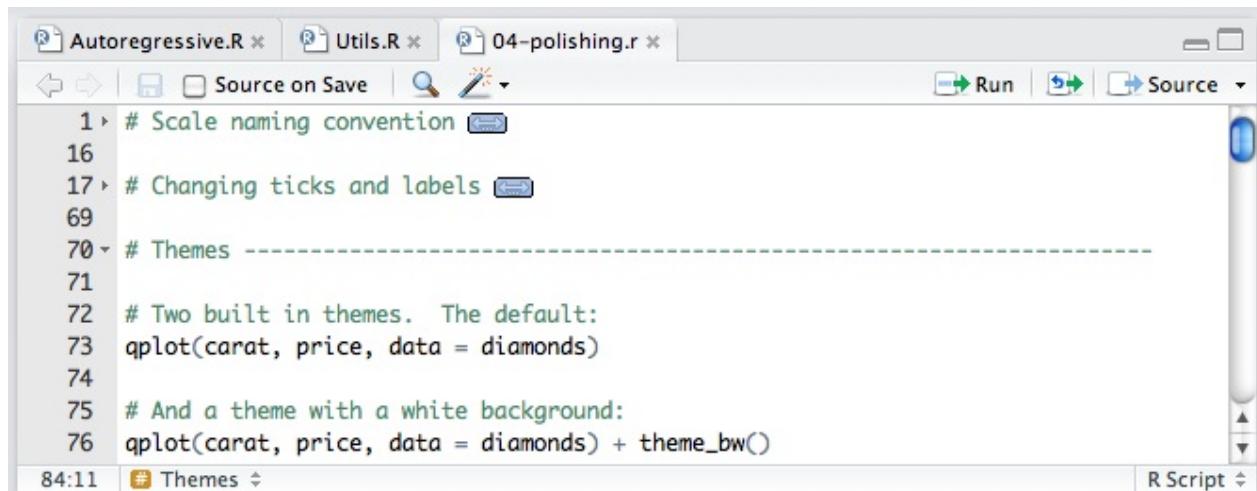


Figura3 - Code folding & Sections en RStudio

Resumen

- Creamos nuevos objetos con el operador de asignación <- .
- Podemos conocer el tipo de un objeto con la función typeof() .
- Todos los objetos pertenecen a una clase.
- Mediante la función class() hallaremos la clase a la que pertenece un objeto.
- Alternativamente, podemos comprobar la clase de un objeto mediante la función is , o mediante una de sus variantes específicas.
- Podemos cambiar la clase de un objeto mediante la función as , o una de sus

variantes específicas.

- R proporciona valores especiales para el ∞ y $-\infty$, NaN (not-a-number) y NA (valores desconocidos).

Estructuras de Datos

Este capítulo resume las estructuras de datos más importantes en R. Las colecciones o conjunto de datos en R se organizan por su dimensión (1º, 2º, o varias dimensiones) y si son homogéneas (todos los objetos deben ser del mismo tipo) o heterogéneas (el contenido puede ser de diferentes tipos). A continuación mostramos los cinco tipos de datos más usados en el análisis de datos:

	Homogénea	Heterogénea
1	Vector atómico	Lista
2	Matriz	Data frame
n	Array	

Tabla 1 Estructuras de datos

Además, analizaremos la sintaxis de R para acceder a las estructuras de datos. Como veremos podemos seleccionar un único elemento o varios elementos, mediante el uso de la notación de índices que proporciona R. Asimismo aprenderemos a elegir elementos por localización dentro de una estructura o por nombre.

La [Tabla 2](#) resume los operadores que aporta R para el acceso a objetos en estructuras de datos.

Sintaxis	Objetos	Descripción
<code>x[i]</code>	Vectores, Listas	Selecciona elementos del objeto <code>x</code> , descritos en <code>i</code> . <code>i</code> puede ser un vector de tipo integer, character (de nombres de los objetos) o lógico. Cuando es usado con listas, devuelve una lista. Cuando es usado en vectores devuelve un vector.
<code>x[[i]]</code>	Listas	Devuelve un único elemento de <code>x</code> que se encuentra en la posición <code>i</code> . <code>i</code> puede ser un vector de tipo integer o character de longitud 1.
<code>x\$n</code>	Listas, Dataframes	Devuelve un objeto con nombre <code>n</code> del objeto <code>x</code> .
<code>[i, j]</code>	Matrices	Devuelve el objeto de la fila <code>i</code> y columna <code>j</code> . <code>i</code> y <code>j</code> pueden ser un vector de tipo integer o character (de nombres de los objetos)

Tabla 2 Notación para acceder estructuras de datos

Objetivos

Después de leer este capítulo, deberíamos:

- Conocer las principales estructuras de datos que proporciona R.
- Ser capaces de crear las distintas colecciones en R.
- Saber manipular las diferentes conjuntos de datos que aporta R.

Vectores

El tipo más básico de estructura de dato en R es el *vector*. El vector es una estructura compuesta de un número de elementos finitos, homogéneos y donde dicha estructura tiene un tamaño fijo. Finito porque existe un último elemento; homogéneo porque todos los elementos son del mismo tipo y tamaño fijo porque el tamaño del vector debe ser conocido en tiempo de ejecución o compilación.

Creación de Vectores

Los vectores atómicos pueden ser creados con la función `c()`, que corresponde a la sigla de *combinar*:

```
vector_double <- c(1, 2.5, 4.5)
# Con el sufijo L, conseguimos un integer en lugar de un double
vector_integer <- c(1L, 6L, 10L)
# Usamos TRUE y FALSE (o T y F) para crear vectores lógicos
vector_logical <- c(TRUE, FALSE, T, F)
vector_character <- c("Hola", "Mundo!")
```

La función `vector()` crea un vector de un tipo y longitud que debemos especificar en el momento de su declaración:

```
vector_double <- vector(mode = "double", length = 3)
vector_integer <- vector(mode = "integer", length = 3)
vector_logical <- vector(mode = "logical", length = 4)
vector_character <- vector(mode = "character", length = 2)
```

Otra posibilidad es hacer uso de las funciones *wrapper* (del inglés, envoltorio) que existen para cada tipo de datos. Las siguientes instrucciones son equivalentes a las anteriores:

```
vector_double <- double(3)
vector_integer <- integer(3)
vector_logical <- logical(4)
vector_character <- character(2)
```

Además, mediante el operador `:` podemos generar sucesiones de números:

```
1:10
## [1] 1 2 3 4 5 6 7 8 9 10
15:11
## [1] 15 14 13 12 11
1:10 - 1
## [1] 0 1 2 3 4 5 6 7 8 9
1:(10 - 1)
## [1] 1 2 3 4 5 6 7 8 9
```

También podemos usar las funciones `seq()` y `rep()` :

```
seq(10) # mismo efecto que 1:10
## [1] 1 2 3 4 5 6 7 8 9 10
seq(3, 10) # mismo efecto que 3:10
## [1] 3 4 5 6 7 8 9 10
seq(1, 10, by = 3) #saltando de 3 en 3
## [1] 1 4 7 10
```

```
rep(1:4, 2) #repetimos 1:4 dos veces
## [1] 1 2 3 4 1 2 3 4
rep(1:4, each = 2) #repetimos 1:4 dos veces, intercalando resultado
## [1] 1 1 2 2 3 3 4 4
```

Longitud

Todos los vectores tienen dos propiedades:

- Un *tipo*, que se puede determinar con la función `typeof()` :

```
typeof(letters)
## [1] "character"
typeof(1:10)
## [1] "integer"
```

- Una *longitud*, que nos dice cuantos elementos contiene el vector. Podemos conocer este valor mediante la función `length()` :

```
v <- c(1, 2, 3)
length(v)
## [1] 3
length(c(TRUE, FALSE, NA))
## [1] 3
```

Una posible fuente de confusión es cuando trabajamos con vectores de tipo character. Con este tipo de vector, la longitud es el número de strings, no el número de caracteres en cada string. Para esto último, utilizaremos la función `nchar()`:

```
alumnos <- c("Juan", "Pepe", "Maria", "Dolores")
length(alumnos)
## [1] 4
nchar(alumnos)
## [1] 4 4 5 7
```

Tipos Fundamentales de Vectores Atómicos

Los cuatro tipos más importantes de vectores atómicos son:

- Logical
- Integer
- Double (conocidos por numeric)
- Character

Cabe mencionar que existen los tipos complex y raw que son raramente utilizados en el análisis de datos, es por eso que no los trataremos en este texto.

Logical

Los vectores lógicos son el tipo más simple de vector atómico puesto que sólo pueden tomar tres posibles valores `TRUE`, `FALSE` y `NA`. Los vectores lógicos usualmente son el resultado de expresiones con los operadores lógicos y de comparación.

```
1:10%%3 == 0
## [1] FALSE FALSE TRUE FALSE FALSE TRUE FALSE FALSE TRUE FALSE
```

Para mas información sobre la sintaxis de los operadores y su precedencia consultar la documentación R:

```
# Sintaxis de los operadores y su precedencia
help("Syntax", "base")
# Operadores lógicos
help("Logic", "base")
```

Numeric

Los vectores de tipo integer y double son en R vectores de tipo numeric. En R, los números son double por defecto. Si queremos un integer, añadiremos la letra `L` después del número:

```
typeof(1)
## [1] "double"
typeof(1L)
## [1] "integer"
1.5
## [1] 1.5
```

Character

Los vectores de tipo character son aquellos en los que cada elemento del vector es un *string* (cadena de caracteres):

```
titulo <- "Ciencia de datos en R"
```

Manipulación de Vectores Atómicos

Ahora que ya conocemos los diferentes tipos de vectores atómicos, pasamos a ver a continuación las diferentes herramientas para trabajar con ellos. En concreto veremos:

- Cómo conocer si un objeto es un tipo específico de vector.
- De qué modo convertir de un tipo a otro, y cuándo sucede de forma automática.
- De qué manera dar nombre a los elementos de un vector.
- Conocer el significado de las operaciones vectorizadas y hacer uso de las mismas.
- Qué sucede cuando trabajamos con vectores de diferentes longitudes.
- Cómo seleccionar partes o elementos de un vector.

Funciones para Comprobar el tipo

En ocasiones queremos realizar diferentes cosas basadas en el tipo de vector. Una opción es usar la función `typeof()`. Otra es usar las funciones que realizan la comprobación de tipo y devuelven `TRUE` o `FALSE`, como `is.character()`, `is.double()`, `is.integer()`, `is.logical()` o, de forma más general mediante `is.atomic()`:

```
vector_integer <- c(1L, 2L, 3L)
typeof(vector_integer)
## [1] "integer"
is.integer(vector_integer)
## [1] TRUE
is.atomic(vector_integer)
## [1] TRUE
```

```
vector_double <- c(1, 2.5, 4.5)
typeof(vector_double)
## [1] "double"
is.double(vector_double)
## [1] TRUE
is.atomic(vector_double)
## [1] TRUE
```

Es importante subrayar que la función `is.numeric()` comprueba si un objeto es de tipo numeric y devuelve `TRUE` tanto para vectores de tipo integer como de tipo double.

```
is.numeric(vector_integer)
## [1] TRUE
is.numeric(vector_double)
## [1] TRUE
```

En la siguiente tabla resumimos las funciones para comprobar el tipo de nuestros vectores:

	logical	integer	double	character	list
is.logical()	X				
is.integer()		X			
is.double()			X		
is.numeric()		X	X		
is.character()				X	
is.atomic()	X	X	X	X	
is.list()					X
is.vector()	X	X	X	X	X

Tabla 2 Funciones comprobación de tipos

Coerción

Todos los elementos de un vector deben ser del mismo tipo, así pues cuando intentemos combinar diferentes tipos estos serán convertidos al tipo más flexible. El orden es el siguiente:

```
logical < integer < double < character
```

Por ejemplo, mezclar un character y un integer producirá un character:

```
v <- c("a", 1)
v
## [1] "a" "1"
typeof(v)
## [1] "character"
class(v)
## [1] "character"
```

Cuando un vector lógico es convertido a un integer o double, `TRUE` es cambiado a 1 y `FALSE` a 0:

```
v <- c(FALSE, FALSE, FALSE)
as.numeric(v)
## [1] 0 0 0
```

La coerción sucede normalmente de forma automática. La mayoría de funciones matemáticas (`+`, `log`, `abs`, etc.) convierten a los tipos double o integer, y la mayoría de operaciones lógicas (`&`, `|`, `any`, etc.) cambian al tipo logical. Si el cambio de un tipo en otro pierde información, R nos lo advertirá mediante un mensaje.

Nombres de los Elementos

Una gran característica de los vectores en R es que podemos asignar a cada elemento un nombre. Etiquetar los elementos hace nuestro código mas legible. Podemos especificar los nombres cuando creamos un vector con la forma `nombre = valor :`

```
c(manzana = 1, platano = 2, kiwi = 3)
## manzana platano    kiwi
##        1         2         3
```

Podemos añadir nombres a los elementos de un vector después de su creación con la ayuda de la función `names()`:

```
frutas <- 1:4
names(frutas) <- c("manzana", "platano", "kiwi")
frutas
## manzana  platano    kiwi    <NA>
##      1        2        3        4
```

Gracias a la función `names()` podemos conocer los nombres de un vector:

```
names(frutas)
## [1] "manzana" "platano" "kiwi"     NA
```

Por último, si un vector no tiene nombres, la función `names()` devuelve `NULL`:

```
names(1:4)
## NULL
```

Operaciones Vectorizadas

La mayoría de las operaciones en R son *vectorizadas*, esto es que un operador o una función actúa en cada elemento de un vector sin la necesidad de que tengamos que escribir una construcción iterativa. Esta característica nos permite escribir un código más eficiente, conciso y mas legible que en otros lenguajes de programación.

El ejemplo mas simple es cuando sumamos dos vectores:

```
v1 <- 1:4
v2 <- 5:8
v3 <- v1 + v2
v3
## [1] 6 8 10 12
```

Cuando usamos dos o mas vectores en una operación, R alinearán los vectores y llevará a cabo una secuencia de operaciones individuales. En el ejemplo anterior, cuando ejecutamos la instrucción `v1 + v2`, R pondrá cada vector en una fila y sumará el primer elemento de `v1` con el primer elemento de `v2`. A continuación, sumará el segundo elemento del vector 1 con el segundo elemento del vector 2, y así sucesivamente, hasta que todos los elementos se han sumado. El resultado será un vector de la misma longitud que los anteriores, como muestra la [Figura 1](#).

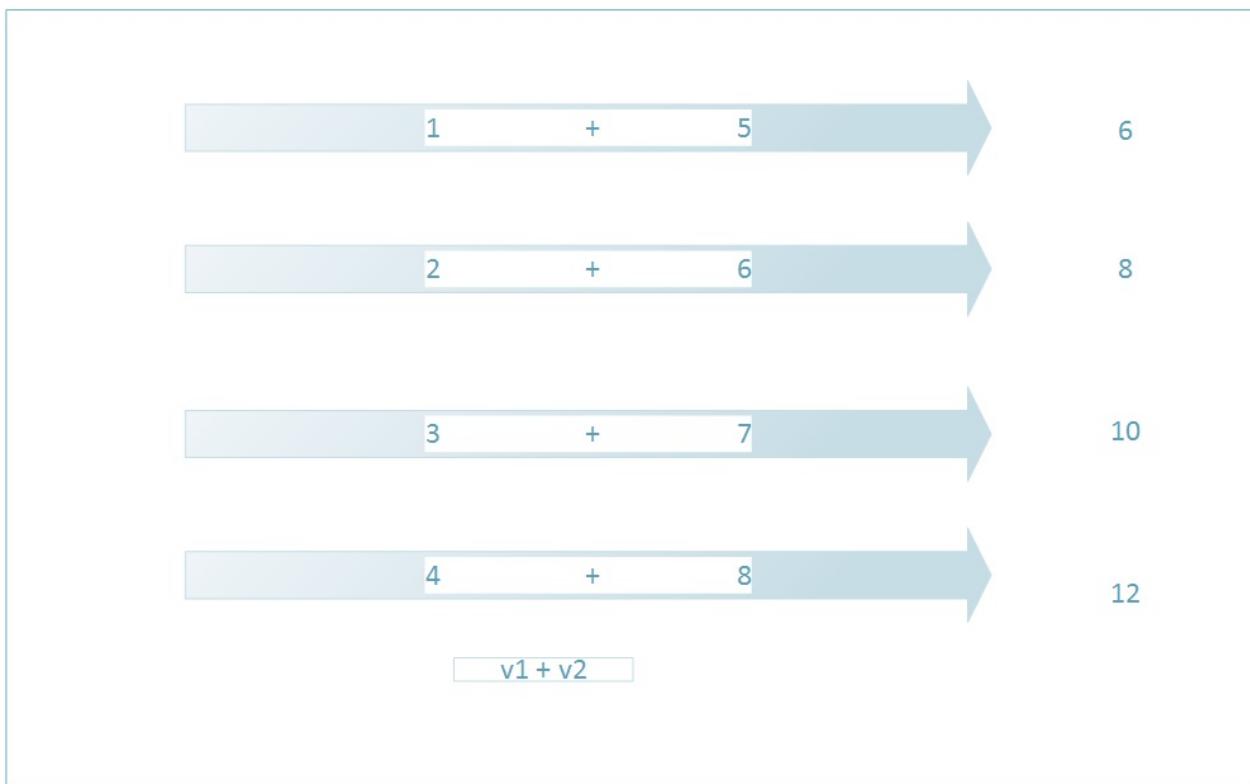


Figura 1 - Operaciones vectorizadas

Sin la vectorización tendríamos que realizar la suma mediante el uso de una estructura iterativa, como por ejemplo:

```
v3 <- numeric(length(v1))
for (i in seq_along(v1)) {
  v3[i] <- v1[i] + v2[i]
}
v3
## [1] 6 8 10 12
```

Otro tipo de operaciones que podemos realizar de forma vectorizada son las comparaciones lógicas. Supongamos que queremos saber que elementos en un vector son mas grandes que 2. Podríamos hacer lo siguiente:

```
v1 <- 1:4
v1 > 2
## [1] FALSE FALSE TRUE TRUE
```

A continuación, mostramos otros ejemplos de operaciones vectorizadas de tipo lógico:

```
v1 <- 1:4
v1 >= 2
## [1] FALSE TRUE TRUE TRUE
v2 < 3
## [1] FALSE FALSE FALSE FALSE
v3 == 8
## [1] FALSE TRUE FALSE FALSE
```

Desde luego, la resta, multiplicación y división son también operaciones vectorizadas:

```
v1 - v2
## [1] -4 -4 -4 -4
v1 * v2
## [1] 5 12 21 32
v1/v2
## [1] 0.2000000 0.3333333 0.4285714 0.5000000
```

Reciclado de Vectores y Repetición

En los ejemplos anteriores hemos realizado operaciones aritméticas con vectores de la misma longitud. No obstante podríamos estar preguntándonos, "¿Qué sucede si intentamos realizar operaciones en vectores de diferente longitud?".

Si intentamos realizar la suma de un único número (escalar) a un vector, entonces el número es sumado a cada elemento en el vector, como muestra la [Figura 2](#):

```
1:5 + 1
## [1] 2 3 4 5 6
```

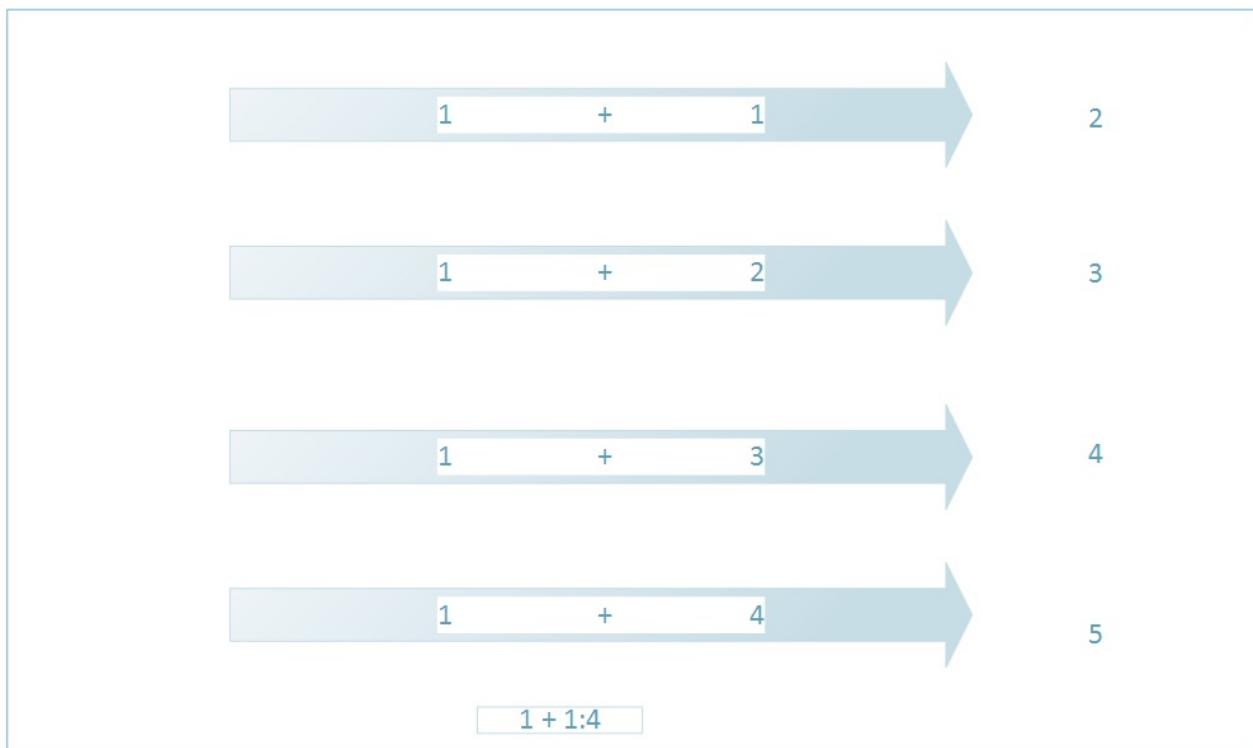


Figura2 - Vectorización con un escalar

Cuando sumamos dos vectores de diferente longitud, R reciclará los elementos del vector más pequeño para que coincida con el más grande, como podemos ver en la Figura 3:

```
1:2 + 1:4  
## [1] 2 4 4 6
```

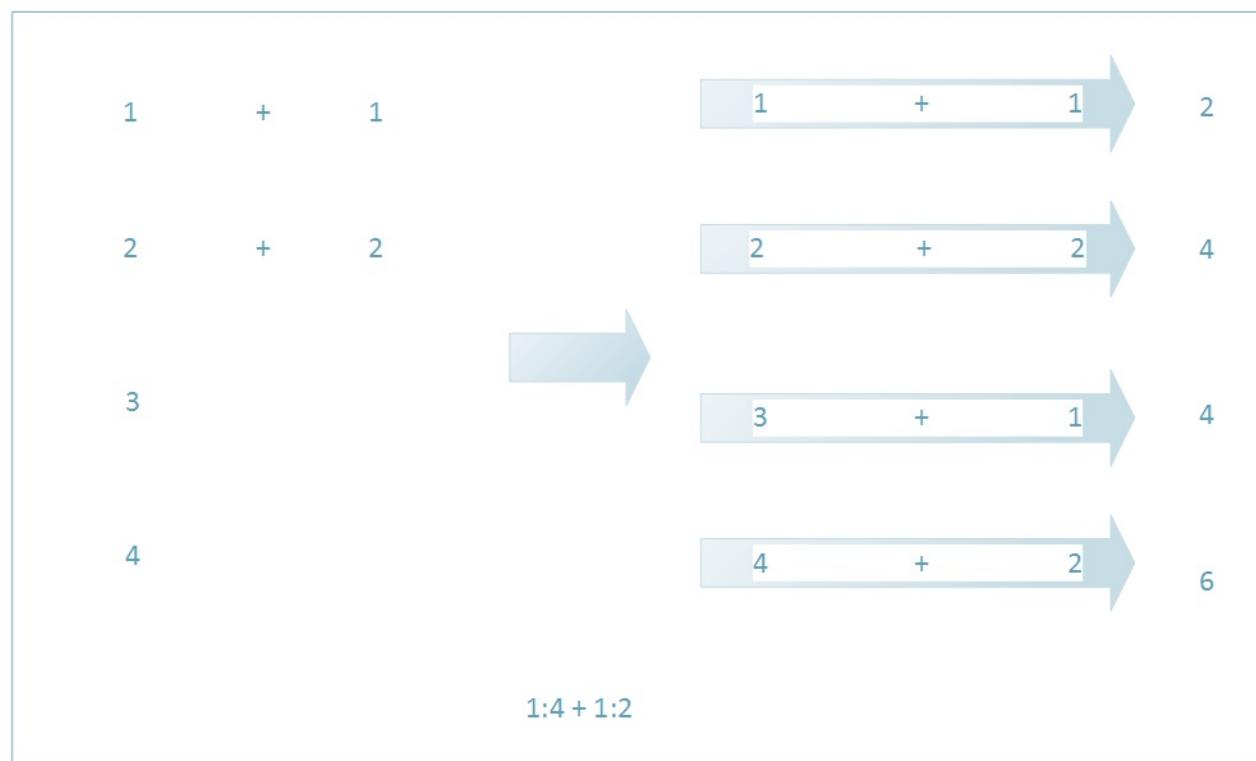


Figura3 - Vectorización vectores diferente longitud

Si la longitud del vector más grande no es múltiple con la longitud del vector más pequeño, R nos lo hará saber mediante un mensaje:

```
1:5 + 1:7
## Warning in 1:5 + 1:7: longer object length is not a multiple of shorter
## object length
## [1] 2 4 6 8 10 7 9
```

Aunque R nos permita realizar operaciones con vectores de diferente longitud, esto no significa que nosotros deberíamos hacerlo. Realizar una suma de un valor escalar a un vector es coherente, pero realizar operaciones con vectores de diferente longitud puede llevarnos a errores. Es por eso, que recomendamos crear explícitamente vectores de la misma longitud antes de operar con ellos.

La función `rep()` es muy útil para esta tarea, permitiéndonos crear un vector con elementos repetidos:

```
rep(1:5, 3)
## [1] 1 2 3 4 5 1 2 3 4 5 1 2 3 4 5
rep(1:5, each = 3)
## [1] 1 1 1 2 2 2 3 3 3 4 4 4 4 5 5 5
rep(1:5, times = 1:5)
## [1] 1 2 2 3 3 3 4 4 4 4 5 5 5 5 5
```

```
rep(1:5, length.out = 7)
## [1] 1 2 3 4 5 1 2
# Alternativamente podemos hacerlo mediante rep_len (desde v3.0.0)
rep_len(1:5, 7)
## [1] 1 2 3 4 5 1 2
```

Selección de Elementos

En ocasiones queremos acceder a una única parte de un vector, o quizá a un único elemento. Esto es conocido como *indexing* (del inglés, indexación) y se realiza mediante el uso de los corchetes `[]`. Existen cuatro maneras diferentes de elegir una parte de un vector:

- Mediante un vector numérico de tipo integer. Los integers deben ser todos positivos, todos negativos, o cero.

Seleccionar los elementos con integers positivos extrae los elementos de las posiciones indicadas:

```
v <- c("uno", "dos", "tres", "cuatro", "cinco")
v[c(3, 2, 5)]
## [1] "tres"  "dos"   "cinco"
```

Repitiendo una posición, podemos obtener un vector de una longitud más grande que el vector original:

```
v[c(1, 1, 5, 5, 5, 2)]
## [1] "uno"    "uno"    "cinco"  "cinco"  "cinco"  "dos"
```

Los valores negativos eliminan los elementos en las posiciones especificadas:

```
v[c(-1, -3, -5)]
## [1] "dos"    "cuatro"
```

No podemos mezclar valores positivos y negativos:

```
v[c(1, -1)]
```

- Por medio de un vector lógico obtenemos todos los valores correspondientes al valor `TRUE`. Este tipo es útil en conjunción con la funciones de comparación:

```
v <- c(10, 3, NA, 5, 8, 1, NA)
# Devuelve todos los valores que no son NA en x
v[!is.na(v)]
## [1] 10 3 5 8 1
```

```
# Todos los valores pares (o desconocidos) en x
v[v%%2 == 0]
## [1] 10 NA 8 NA
```

- Si hemos dado nombres a los elementos de nuestro vector, podemos seleccionar sus elementos con un vector de tipo character:

```
frutas <- c(manzana = 1, platano = 2, kiwi = 3, pera = 4, naranja = 5)
frutas[c("platano", "naranja")]
## platano naranja
##       2      5
```

- Mediante `v[]`, obtendremos el vector completo:

```
v <- c(10, 3, NA, 5, 8, 1, NA)
v[]
## [1] 10 3 NA 5 8 1 NA
```

Esta notación no es muy útil para acceder a vectores, sin embargo nos será de gran ayuda en el acceso a **matrices** (y cualquier tipo de estructura multidimensional) puesto que nos permite seleccionar todas las filas o columnas. Por ejemplo, si x es $2D$, `v[1,]` selecciona la primera fila y todas las columnas, y `v[, -1]` recupera todas las filas y columnas excepto la primera.

Resumen

- Los vectores tienen una longitud que podemos conocer o definir mediante la función `length()`.
- La función `seq()` y sus variantes nos permite crear sucesiones de números.
- Podemos dar nombre a los elementos de un vector en el momento de su creación o una vez creado mediante la función `names()`.
- Podemos acceder a los elementos de un vector mediante los `[]` y un índice. La función `rep()` nos permite crear vectores con elementos repetidos.

Matrices

Una *matriz* es una extensión de un vector a dos dimensiones, lo que implica que dispone del atributo *dimension*. El atributo dimensión es en si mismo un vector de longitud 2 (numero de filas, numero de columnas). Una matriz se utiliza para representar datos de un único tipo en dos dimensiones.

Creación de Matrices

Para crear matrices utilizaremos la función `matrix()`, la sintaxis es la siguiente

```
> str(matrix)
```

```
function (data = NA, nrow = 1, ncol = 1, byrow = FALSE, dimnames = NULL)
```

A continuación mostramos la descripción de los argumentos:

- **data** es el vector que contiene los elementos que formaran parte de la matriz.
- **nrow** es el número de filas.
- **ncol** es el número de columnas.
- **byrow** es un valor lógico. Si es `TRUE` el vector que pasamos será ordenado por filas.
- **dimnames** nombres asignado a filas y columnas.

Seguidamente se muestra un ejemplo de creación de una matriz:

```
> matriz <- matrix(1:12, nrow = 4)
> matriz
```

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

A partir de un vector, si le añadimos el atributo dimensión podemos obtener una matriz:

```
> m <- 1:12
> m
```

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

```
> dim(m) <- c(4, 3)
> m
```

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

Cuando creamos una matriz, los valores que pasamos son ordenados por columnas. Pero también es posible llenar la matriz por filas especificando el argumento `byrow = TRUE` :

```
> matriz <- matrix(1:12, nrow = 4, byrow = TRUE)
> matriz
```

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

En el siguiente ejemplo hacemos uso del argumento `dimnames` para dar nombre a las filas y columnas:

```
> automoviles <- matrix(
+     1:12,
+     nrow = 4,
+     byrow = TRUE,
+     dimnames = list(
+         c("Blanco", "Rojo", "Negro", "Gris"),
+         c("Toyota", "Audi", "Nissan")
+     )
+ )
> automoviles
```

	Toyota	Audi	Nissan
Blanco	1	2	3
Rojo	4	5	6
Negro	7	8	9
Gris	10	11	12

Mediante las funciones `cbind()` y `rbind()` es posible crear matrices por columnas o por filas a partir de dos vectores de la misma longitud:

```
> v1 <- c(1, 2, 3)
> v2 <- c(4, 5, 6)
> m1 <- cbind(v1, v2)
> m1
```

```
v1 v2
[1,] 1 4
[2,] 2 5
[3,] 3 6
```

```
> v1 <- c(1, 2, 3)
> v2 <- c(4, 5, 6)
> m1 <- rbind(v1, v2)
> m1
```

```
[,1] [,2] [,3]
v1     1     2     3
v2     4     5     6
```

Filas, Columnas y Dimensión

La función `dim()` devuelve un vector de integers con la dimensión del objeto:

```
> dim(automoviles)
```

```
[1] 4 3
```

Además con las funciones `nrow()` y `ncol()` podemos conocer el número de filas y columnas, respectivamente:

```
> nrow(automoviles)
```

```
[1] 4
```

```
> ncol(automoviles)
```

```
[1] 3
```

La función `length()` que hemos visto con anterioridad en los vectores, también funciona en matrices. Cuando trabajamos con matrices; no obstante, devuelve el producto de cada una de las dimensiones:

```
> length(automoviles)
```

```
[1] 12
```

Nombres de las Filas, Columnas y Dimensiones

Del mismo modo que los vectores poseen el atributo `names` para sus elementos, las matrices disponen de `rownames` y `colnames` para las filas y columnas.

```
> colores <- rownames(automoviles)  
> colores
```

```
[1] "Blanco" "Rojo"   "Negro"  "Gris"
```

```
> marcas <- colnames(automoviles)  
> marcas
```

```
[1] "Toyota" "Audi"    "Nissan"
```

Por medio de la función `dimnames()` obtendremos una [lista](#) que contiene dos vectores con los atributos `rownames` y `colnames`:

```
> dimnames(automoviles)
```

```
[[1]]
[1] "Blanco" "Rojo"   "Negro"  "Gris"

[[2]]
[1] "Toyota" "Audi"   "Nissan"
```

Operaciones con Matrices

La función `diag()` extrae la diagonal principal de una matriz:

```
> A <- matrix(c(1, 2, 3, 4, 5, 6, 7, 8, 9), nrow = 3, ncol = 3, byrow = TRUE)
> A
```

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

```
> diag(A)
```

```
[1] 1 5 9
```

Además, `diag()` nos permite crear matrices diagonales:

```
> diag(c(1, 2, 3, 4))
```

```
 [,1] [,2] [,3] [,4]
[1,]    1    0    0    0
[2,]    0    2    0    0
[3,]    0    0    3    0
[4,]    0    0    0    4
```

La matriz identidad es muy fácil de crear en R. Por ejemplo, la matriz identidad de dimensión 4 es:

```
> Id4 = diag(1, nrow = 4)
> Id4
```

```
[,1] [,2] [,3] [,4]
[1,] 1 0 0 0
[2,] 0 1 0 0
[3,] 0 0 1 0
[4,] 0 0 0 1
```

Hay que tener cierto cuidado con los operadores aritméticos básicos (`+`, `-`, `*`). Si se suman una matriz y una constante, el efecto es que dicha constante se suma a todos los elementos de la matriz. Lo mismo ocurre con la diferencia, la multiplicación y la división:

```
> M = matrix(nrow=2,c(1,2,3, 4),byrow = FALSE)
> M
```

```
[,1] [,2]
[1,] 1 3
[2,] 2 4
```

```
> M + 2
```

```
[,1] [,2]
[1,] 3 5
[2,] 4 6
```

Asimismo, si a una matriz se le suma un vector cuya longitud sea igual al número de filas de la matriz, se obtiene como resultado una nueva matriz cuyas columnas son la suma de las columnas de la matriz original más dicho vector.

```
> v = c(3,4)
> M + v
```

```
[,1] [,2]
[1,] 4 6
[2,] 6 8
```

La suma o resta de matrices de la misma dimensión se realiza con los operadores `+` y `-`; el producto de matrices (siempre que sean compatibles) se realiza con el operador `%%*`:

```
> M + M
```

```
[,1] [,2]
[1,]    2    6
[2,]    4    8
```

```
> M - M
```

```
[,1] [,2]
[1,]    0    0
[2,]    0    0
```

```
> M%*%M
```

```
[,1] [,2]
[1,]    7   15
[2,]   10   22
```

Una fuente de posibles errores en el cálculo matricial, cuando se utilizan matrices de la misma dimensión, es utilizar los operadores `*` y `/` ya que multiplican (o dividen) las matrices término a término:

```
> M * M
```

```
[,1] [,2]
[1,]    1    9
[2,]    4   16
```

```
> M / M
```

```
[,1] [,2]
[1,]    1    1
[2,]    1    1
```

La traspuesta de una matriz se calcula simplemente con la función `t()`:

```
> M
```

```
[,1] [,2]
[1,]    1    3
[2,]    2    4
```

```
> t(M)
```

```
[,1] [,2]
[1,]    1    2
[2,]    3    4
```

El determinante de una matriz cuadrada se calcula mediante la función `det()`:

```
> det(M)
```

```
[1] -2
```

La función `solve()` permite obtener la inversa de una matriz cuando sólo se le pasa un argumento:

```
> solve(M)
```

```
[,1] [,2]
[1,]   -2  1.5
[2,]    1 -0.5
```

Además, con la función `solve()` podemos resolver sistemas de ecuaciones lineales. Por ejemplo, si disponemos del siguiente sistema de ecuaciones:

$$Ax = b \begin{cases} 3x + 2y = 5 \\ x - y = 0 \end{cases}$$

que en forma matricial puede expresarse como

$$\begin{pmatrix} 3 & 2 \\ 1 & -1 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 5 \\ 0 \end{pmatrix}$$

Podemos resolver el sistema de ecuaciones en R, del siguiente modo:

```
> A <- matrix(c(3, 2, 1, -1), ncol = 2, byrow = TRUE)
> b <- c(5, 0)
> solve(A, b)
```

```
[1] 1 1
```

Selección de Elementos

Los elementos de una matriz están indexados con dos índices lo cual hace que resulte fácil acceder a los elementos y trabajar con ellos si lo que nos interesa es sólo una parte de la información contenida en una matriz y no la matriz entera, esto se logra con el operador de indexación `[i, j]` donde i es el elemento fila y j es el elemento columna.

Siguiendo con el ejemplo anterior, si quisiéramos seleccionar el número de automóviles blancos correspondiente a la marca Audi podríamos hacerlo de dos maneras:

- Escribiendo el nombre de la matriz y entre corchetes los nombres de la fila y columnas entre comillas:

```
> automoviles["Blanco", "Audi"]
```

```
[1] 2
```

- Alternativamente, podemos utilizar la notación de índices:

```
> automoviles[1, 2]
```

```
[1] 2
```

También podemos seleccionar columnas y filas enteras, de manera que si queremos seleccionar todos los automóviles blancos lo haríamos del siguiente modo:

```
> automoviles[1, ]
```

Toyota	Audi	Nissan
1	2	3

```
> # otra forma de hacerlo es  
> automoviles["Blanco", ]
```

```
Toyota  Audi Nissan
      1      2      3
```

Agregar Filas y Columnas

Podemos emplear las funciones `cbind()` y `rbind()` para agregar filas y columnas a una matriz que hemos creado con anterioridad:

```
> # Añadimos una nueva fila a la matriz
> verde <- c(8, 5, 7)
> automoviles <- rbind(automoviles, verde)
> automoviles
```

	Toyota	Audi	Nissan
Blanco	1	2	3
Rojo	4	5	6
Negro	7	8	9
Gris	10	11	12
verde	8	5	7

```
> # Añadimos una nueva columna
> ford <- c(2, 7, 3, 5, 9)
> automoviles <- cbind(automoviles, ford)
> automoviles
```

	Toyota	Audi	Nissan	ford
Blanco	1	2	3	2
Rojo	4	5	6	7
Negro	7	8	9	3
Gris	10	11	12	5
verde	8	5	7	9

Eliminar Filas y Columnas

Para eliminar filas utilizaremos la notación `[-i,]`, de forma similar para eliminar columnas utilizaremos la notación `[, -j]`. A modo de ejemplo, vamos a eliminar la fila y columna que hemos añadido en el apartado anterior:

```
> #Eliminando la fila verde
> automoviles[-5, ]
```

	Toyota	Audi	Nissan	ford
Blanco	1	2	3	2
Rojo	4	5	6	7
Negro	7	8	9	3
Gris	10	11	12	5

```
> # Eliminando columna ford
> automoviles[, -4]
```

	Toyota	Audi	Nissan
Blanco	1	2	3
Rojo	4	5	6
Negro	7	8	9
Gris	10	11	12
verde	8	5	7

Resumen

- La función `dim()` devuelve un vector de integers con la dimensión del objeto.
- Además con las funciones `nrow()` y `ncol()` podemos conocer el número de filas y columnas.
- Por medio de la función `dimnames()` obtendremos una lista que contiene dos vectores con los atributos `rownames` y `colnames`.
- Podemos seleccionar elementos de un vector usando la notación `[i,j]`.
- Con la ayuda de las funciones `cbind()` y `rbind()` podemos agregar filas y columnas.
- Para eliminar filas utilizaremos la notación `[-i,]`.
- De forma similar para eliminar columnas utilizaremos la notación `[, -j]`.

Arrays

Un *array* es una extensión de un vector a más de dos dimensiones. Los arrays se emplean para representar datos multidimensionales de un único tipo. Los arrays son raramente utilizados en el análisis de datos, por este motivo no profundizaremos en su estudio en este texto.

Creación de Arrays

Para crear un *array* utilizaremos la función `array()`, a la que pasaremos un vector atómico con los valores y un vector de dimensiones. Opcionalmente, podemos proporcionar nombres para cada dimensión:

```
array_3_D <- array(1:24, dim = c(4, 3, 2), dimnames = list(c("uno", "dos", "tres",
  "cuatro"), c("five", "six", "seven"), c("un", "deux")))
array_3_D
## , , un
##
##      five six seven
## uno      1   5    9
## dos      2   6   10
## tres     3   7   11
## cuatro   4   8   12
##
## , , deux
##
##      five six seven
## uno     13  17   21
## dos     14  18   22
## tres    15  19   23
## cuatro  16  20   24
```

Podemos comprobar si un objeto es un array mediante la función `is.array()`:

```
is.array(array_3_D)
## [1] TRUE
```

Finalmente, podemos conocer su dimensión con la ayuda de la función `dim()`:

```
dim(array_3_D)
## [1] 4 3 2
```


Listas

Podemos entender una *lista* como un contenedor de objetos que pueden ser de cualquier clase: números, vectores, matrices, [funciones](#), [data.frames](#), incluso otras listas. Una lista puede contener a la vez varios de estos objetos, que pueden ser además de distintas dimensiones.

Creación de listas

Podemos crear listas con la función `list()`, que acepta un número arbitrario de argumentos. Los elementos de la lista pueden ser cualquier tipo de objeto:

```
lista <- list(1:3, "Ruben", pi, list(c(-1, -2), -5))
lista
## [[1]]
## [1] 1 2 3
##
## [[2]]
## [1] "Ruben"
##
## [[3]]
## [1] 3.141593
##
## [[4]]
## [[4]][[1]]
## [1] -1 -2
##
## [[4]][[2]]
## [1] -5
```

Como con los vectores, podemos dar nombre a los elementos en su construcción, o posteriormente con la ayuda de la función `names()`:

```
names(lista) <- c("a", "b", "c", "d")
lista
## $a
## [1] 1 2 3
##
## $b
## [1] "Ruben"
##
## $c
## [1] 3.141593
##
## $d
## $d[[1]]
## [1] -1 -2
##
## $d[[2]]
## [1] -5
```

```
la_misma_lista <- list(a = 1:3, b = "Ruben", c = pi, d = list(c(-1, -2), -5))
la_misma_lista
## $a
## [1] 1 2 3
##
## $b
## [1] "Ruben"
##
## $c
## [1] 3.141593
##
## $d
## $d[[1]]
## [1] -1 -2
##
## $d[[2]]
## [1] -5
```

Un herramienta muy útil para el trabajo con listas es la función `str()` que nos muestra su estructura:

```
str(lista)
## List of 4
## $ a: int [1:3] 1 2 3
## $ b: chr "Ruben"
## $ c: num 3.14
## $ d:List of 2
##   ..$ : num [1:2] -1 -2
##   ..$ : num -5
```

Selección de Elementos

Disponemos de tres métodos para seleccionar elementos de una lista, que examinaremos a partir de `lista` :

```
lista <- list(a = 1:3, b = "Ruben", c = pi, d = list(c(-1, -2), -5))
```

- La notación `[]` extrae una sublista. El resultado será siempre una lista:

```
str(lista[1:2])
## List of 2
## $ a: int [1:3] 1 2 3
## $ b: chr "Ruben"
```

```
str(lista[4])
## List of 1
## $ d:List of 2
##   ..$ : num [1:2] -1 -2
##   ..$ : num -5
```

Como con los vectores, podemos seleccionar elementos con un vector de tipo `logical` , `integer` o `character` .

- La notación `[[[]]]` extrae un único componente de la lista. Esto es, elimina un nivel en la jerarquía de la lista:

```
str(lista[[1]])
##  int [1:3] 1 2 3
```

```
str(lista[[4]])
## List of 2
## $ : num [1:2] -1 -2
## $ : num -5
```

```
str(lista[[4]][1])
## List of 1
## $ : num [1:2] -1 -2
```

```
str(lista[[4]][[1]])
##  num [1:2] -1 -2
```

- El operador `$` extrae elementos de una lista por medio de su nombre. El funcionamiento es el mismo que con el operador `[[[]]]` excepto que no tenemos que utilizar comillas (`" "`):

```
str(lista$a)
##  int [1:3] 1 2 3
```

```
str(lista[["a"]])
##  int [1:3] 1 2 3
```

La distinción entre `[]` y `[[[]]]` es importante en las listas, puesto que `[[[]]]` navega jerárquicamente por la lista, mientras que `[]` devuelve una sublista. A continuación mostramos una representación visual de las operaciones realizadas en el código anterior:

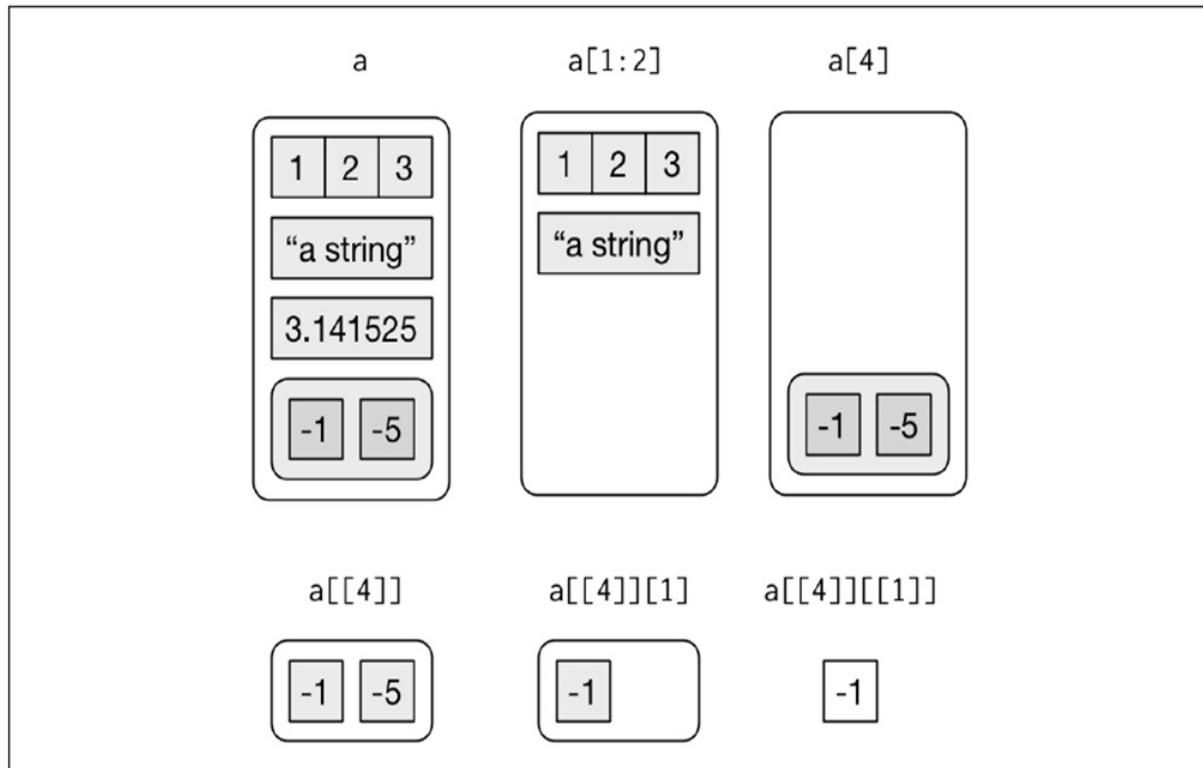


Figura 1 - Selección elementos de una lista, visualmente

Resumen

- Creamos listas con la función `list()`.
- Con la ayuda de la función `names()` podemos consultar o cambiar los nombres de los

elementos.

- Una función útil para el trabajo con listas es `str()` que nos muestra su estructura.
- Hemos analizado los diferentes operadores para seleccionar elementos de una lista.

Data frames

Los *data frames* se utilizan en R para almacenar datos en forma de hoja de datos. Cada fila de la hoja de datos corresponde a una observación o valor de una instancia, mientras que cada columna corresponde a un vector que contiene los datos de una variable.

Como podemos observar en el ejemplo, cada observación, indicado en la primera columna con un número, tiene ciertas características que son representadas en las tres columnas restantes. Cada columna consiste de valores del mismo tipo, puesto que se corresponden a vectores: es por eso, que la columna `breaks` contiene valores de tipo `double`, mientras que las columnas `wool` y `tension` contienen caracteres que son almacenados como `factor`.

```
head(warpbreaks)
##   breaks wool tension
## 1     26    A      L
## 2     30    A      L
## 3     54    A      L
## 4     25    A      L
## 5     70    A      L
## 6     52    A      L
```

La estructura de un `data.frame` es muy similar a la de una matriz. La diferencia es que las filas de un `data.frame` pueden contener valores de diferentes tipos de datos.

Los `data.frame` también tienen similitud con las listas, puesto que son básicamente colecciones de elementos. Sin embargo, el `data.frame` es una lista que únicamente contiene vectores de la misma longitud. Por lo tanto, podemos considerarlo un tipo especial de lista y en el podemos acceder a sus elementos del mismo modo que lo hacemos en las matrices o las listas.

Creación de un Data Frame

Podemos crear data frames con la función `data.frame()`:

```
# Creamos vectores con los valores
nombre <- c("Juan", "Margarita", "Ruben", "Daniel")
apellido <- c("Sanchez", "Garcia", "Sancho", "Alfara")
fecha_nacimiento <- c("1976-06-14", "1974-05-07", "1958-12-25", "1983-09-19")
sexo <- c("HOMBRE", "MUJER", "HOMBRE", "HOMBRE")
nro_hijos <- c(1, 2, 3, 4)
```

```
# Creamos un dataframe con la ayuda de data.frame()
censo <- data.frame(nombre, apellido, fecha_nacimiento, sexo, nro_hijos)
censo
##      nombre apellido fecha_nacimiento   sexo nro_hijos
## 1      Juan    Sanchez     1976-06-14 HOMBRE      1
## 2 Margarita    Garcia     1974-05-07 MUJER      2
## 3     Ruben    Sancho     1958-12-25 HOMBRE      3
## 4    Daniel    Alfara     1983-09-19 HOMBRE      4
```

Recordemos que los data frames requieren que las variables sean de la misma longitud. Por este motivo, tenemos que asegurarnos que el número de argumentos pasados a la función `c()` sea el mismo. Además, debemos asegurarnos que las cadenas de caracteres están entre `""`.

Además, observemos que cuando hacemos uso de la función `data.frame()`, las variables de tipo `character` son importadas como variables categóricas, que en R son representadas como `factor`.

Si estamos interesados en inspeccionar las primeras o últimas líneas, podemos hacer uso de las funciones `head()` y `tail()`, respectivamente:

```
head(censo)
##      nombre apellido fecha_nacimiento   sexo nro_hijos
## 1      Juan    Sanchez     1976-06-14 HOMBRE      1
## 2 Margarita    Garcia     1974-05-07 MUJER      2
## 3     Ruben    Sancho     1958-12-25 HOMBRE      3
## 4    Daniel    Alfara     1983-09-19 HOMBRE      4
```

Por otro lado, podemos usar la función `str()` para conocer la estructura del data frame:

```
str(censo)
## 'data.frame': 4 obs. of 5 variables:
## $ nombre : Factor w/ 4 levels "Daniel","Juan",...: 2 3 4 1
## $ apellido : Factor w/ 4 levels "Alfara","Garcia",...: 3 2 4 1
## $ fecha_nacimiento: Factor w/ 4 levels "1958-12-25","1974-05-07",...: 3 2 1 4
## $ sexo : Factor w/ 2 levels "HOMBRE","MUJER": 1 2 1 1
## $ nro_hijos : num 1 2 3 4
```

Como podemos observar, las variables `nombre`, `apellido`, `fecha_nacimiento` y `sexo` de `censo` son tratadas como factors. Para evitar este comportamiento podemos hacer lo siguiente:

- Para las variables `nombre` y `apellido` podemos hacer uso de la función `I()` en la definición de los vectores para que las variables no sean tratadas como `factor`. Esta función inhibe la interpretación de sus argumentos.

- La variable `sexo` podemos dejarla como `factor`, puesto que se trata de una cantidad limitada de posibles valores.
- Para que la variable `fecha_nacimiento` sea registrada como fecha, podemos hacer uso de la función `as.Date()`:

```
# Creamos vectores con los valores y el tipo de dato deseado
nombre <- I(c("Juan", "Margarita", "Ruben", "Daniel"))
apellido <- I(c("Sanchez", "Garcia", "Sancho", "Alfara"))
fecha_nacimiento <- as.Date(c("1976-06-14", "1974-05-07", "1958-12-25", "1983-09-19"))
sexo <- c("HOMBRE", "MUJER", "HOMBRE", "HOMBRE")
nro_hijos <- c(1, 2, 3, 4)
```

```
# Creamos un dataframe con la ayuda de data.frame()
censo <- data.frame(nombre, apellido, fecha_nacimiento, sexo, nro_hijos)
str(censo)
## 'data.frame': 4 obs. of 5 variables:
## $ nombre :Class 'AsIs' chr [1:4] "Juan" "Margarita" "Ruben" "Daniel"
## $ apellido :Class 'AsIs' chr [1:4] "Sanchez" "Garcia" "Sancho" "Alfara"
## $ fecha_nacimiento: Date, format: "1976-06-14" "1974-05-07" ...
## $ sexo : Factor w/ 2 levels "HOMBRE","MUJER": 1 2 1 1
## $ nro_hijos : num 1 2 3 4
```

Filas, Columnas y Dimensión

Como las matrices, el `data.frame` dispone del atributo dimensión determinado por el número de filas y columnas. Para comprobar el número de filas y columnas de `censo`, podemos hacer uso de la función `dim()`:

```
# Devuelve el número de filas y columnas
dim(censo)
## [1] 4 5
# Recupera el número de filas
dim(censo)[1]
## [1] 4
# Recupera el número de columnas
dim(censo)[2]
## [1] 5
```

También podemos recuperar el número de filas y columnas en `censo` con la ayuda de las funciones `nrow()` y `ncol()`:

```
# Usamos `nrow()` para recuperar el número de filas
nrow(censo)
## [1] 4
# Usamos `ncol()` para recuperar el número de columnas
ncol(censo)
## [1] 5
# Usamos `length()` para recuperar el número de columnas
length(censo)
## [1] 5
```

Observemos que, puesto que la estructura de un `data.frame` es similar a una lista, podemos hacer uso de la función `length()` para recuperar el número de columnas. No obstante, para que nuestro código sea comprensible, recomendamos evitar el uso de esta función y en su lugar utilizar la función `ncol()`.

Nombre de las Filas y Columnas

Los data frames también cuentan con el atributo `names`, que nos indica los nombres de las variables que hemos definido. En otras palabras, podemos ver y establecer una cabecera.

```
# Listamos los nombres de las variables (cabecera)
names(censo)
## [1] "nombre"           "apellido"          "fecha_nacimiento"
## [4] "sexo"             "nro_hijos"
```

Para cambiar los nombres, podemos hacer uso de la función `names()`:

```
# Asignamos diferentes nombres a las columnas de `censo`
names(censo) <- c("Nombre", "Apellido", "Fecha_Nacimiento", "Sexo", "Numero_Hijos")
names(censo)
## [1] "Nombre"           "Apellido"          "Fecha_Nacimiento"
## [4] "Sexo"             "Numero_Hijos"
```

También podemos cambiar los nombres de las filas y columnas con la ayuda de las funciones `rownames()` y `colnames()`, respectivamente:

```
# Asignamos diferentes nombres a las columnas de `censo`
colnames(censo) <- c("Nombre", "Apellido", "Fecha_Nacimiento", "Sexo", "Numero_Hijos")
rownames(censo) <- c("ID1", "ID2", "ID3", "ID4")
censo
##           Nombre Apellido Fecha_Nacimiento Sexo Numero_Hijos
## ID1        Juan   Sanchez      1976-06-14 HOMBRE       1
## ID2 Margarita   Garcia      1974-05-07 MUJER       2
## ID3     Ruben   Sancho      1958-12-25 HOMBRE       3
## ID4    Daniel   Alfara      1983-09-19 HOMBRE       4
```

Observemos que, la función `names()` devuelve el mismo valor que `colnames()`. Para que nuestro código sea inteligible, recomendamos el uso de la función `colnames()`.

Selección de Elementos

El acceso a los elementos que se encuentran en un `data.frame` es muy similar al acceso a los datos de una matriz que ya vimos en la [sección anterior](#). Así por ejemplo, si queremos ver sólo los datos de los sujetos (filas) 2 a 4, escribiríamos:

```
censo[2:4, ]
##           Nombre Apellido Fecha_Nacimiento Sexo Numero_Hijos
## ID2 Margarita   Garcia      1974-05-07 MUJER       2
## ID3     Ruben   Sancho      1958-12-25 HOMBRE       3
## ID4    Daniel   Alfara      1983-09-19 HOMBRE       4
```

Si queremos acceder a la variable `nombre` (primera columna), podemos tratar a `censo` igual que si fuese una matriz:

```
censo[, 1]
## [1] "Juan"      "Margarita" "Ruben"     "Daniel"
```

Aunque también podemos referirnos a la columna por su nombre:

```
censo$Nombre
## [1] "Juan"      "Margarita" "Ruben"     "Daniel"
```

Nótese que en este caso hemos de utilizar el nombre del `data.frame` `censo` seguido del operador `$` y del nombre de la variable que nos interesa (`Nombre`). De manera equivalente, la selección de esa variable puede realizarse mediante:

```
censo[, "Nombre"]
## [1] "Juan"      "Margarita" "Ruben"     "Daniel"
```

o poniendo el nombre de la variable entre dobles corchetes y entre comillas:

```
censo[["Nombre"]]
## [1] "Juan"      "Margarita" "Ruben"     "Daniel"
```

Attach y Detach

La notación `$` para el acceso a elementos de un `data.frame` puede hacerse engorroso cuando hemos de escribir constantemente el nombre del `data.frame` (en particular si éste es muy largo). Imaginemos, por ejemplo, que para el conjunto de datos `censo` deseamos construir tablas de frecuencias de la variable `nro_hijos`, y que además queremos calcular la media del número de hijos. La sintaxis a utilizar sería la siguiente:

```
# Calculamos la tabla de frecuencias
table(censo$Sexo)
##
## HOMBRE MUJER
##      3      1
# Diagrama de barras variable `sexo`
barplot(table(censo$Sexo))
```



Figura1 - Diagrama de barras variable `sexo`

```
# Calculamos la media de `nro_hijos`
mean(censo$Numero_Hijos)
## [1] 2.5
```

```
# Calculamos la mediana de `nro_hijos`
median(censo$Numero_Hijos)
## [1] 2.5
```

```
# Calculamos la varianza de `nro_hijos`
var(censo$Numero_Hijos)
## [1] 1.666667
```

Obviamente, escribir tantas veces `censo` resulta tedioso, al margen de que se multiplica el riesgo de cometer errores en la redacción de los comandos. Para evitar este problema podemos hacer uso de la función `attach()`, cuyo objetivo consiste básicamente en

"enganchar" el contenido del dataframe al entorno donde R busca los nombres de variable; de esta forma se puede acceder directamente a las variables del dataframe por su nombre:

```
# Attach el dataframe `censo`
attach(censo)
# Calculamos distribución frecuencias absolutas
cbind(table(sexo))
##      [,1]
## HOMBRE    3
## MUJER    1
```

```
# Diagrama de barras de `nro_hijos`
barplot(table(sexo))
```



Figura2 - Diagramas de barras variable `sexo`

```
# Calculamos la media de `nro_hijos`
mean(nro_hijos)
## [1] 2.5
```

```
# Calculamos la mediana de `nro_hijos`
median(nro_hijos)
## [1] 2.5
```

```
# Calculamos la varianza de `nro_hijos`
var(nro_hijos)
## [1] 1.666667
```

```
# Detach el dataframe `censo`
detach(censo)
```

Una vez que hayamos acabado nuestra tarea “desenganchamos” el dataframe con la función `detach()`.

Eliminar Columnas y Filas

Si deseamos eliminar valores o columnas enteras asignaremos el valor `NULL` a la unidad deseada:

```
# Creamos una copia de `censo`
copia_censo <- censo

# Asignamos el valor `NULL` al valor en [1, 3]
copia_censo[1, 3] <- NULL

# Asignamos `NULL` a la variable `nro_hijos`
copia_censo$nro_hijos <- NULL

# Mostramos por pantalla valor en [1, 3]

valor_eliminado <- copia_censo$Numero_Hijos
valor_eliminado
## [1] 1 2 3 4

# Mostramos por pantalla variables `copia_censo`
names(copia_censo)
## [1] "Nombre"          "Apellido"        "Fecha_Nacimiento"
## [4] "Sexo"            "Numero_Hijos"
```

Para eliminar filas definiremos un vector de tipo lógico en el que indicaremos para cada fila si será eliminada o no:

```
# Definimos las filas a conservar
filas_a_conservar <- c(TRUE, FALSE, TRUE, FALSE)

# Obtenemos un subconjunto de `censo` con las filas a conservar
subconjunto_censo <- censo[filas_a_conservar, ]

# Mostramos por pantalla `subconjunto_censo`
subconjunto_censo
##   nombre apellido fecha_nacimiento sexo nro_hijos
## 1 Juan Sanchez 1976-06-14 HOMBRE      1
## 3 Ruben Sancho 1958-12-25 HOMBRE      3
```

Otro modo de eliminar las filas es haciendo lo contrario añadiendo el operador lógico de negación (`!`), a modo de ejemplo utilizaremos el código anterior:

```
# Definimos las filas a conservar
filas_a_conservar <- c(TRUE, FALSE, TRUE, FALSE)

# Obtenemos un subconjunto de `censo` con las filas eliminadas
subconjunto_censo <- censo[!filas_a_conservar, ]

# Mostramos por pantalla `subconjunto_censo`
subconjunto_censo
##     nombre apellido fecha_nacimiento sexo nro_hijos
## 2 Margarita Garcia      1974-05-07 MUJER        2
## 4 Daniel   Alfara     1983-09-19 HOMBRE        4
```

Por último, destacar que podemos definir condiciones lógicas para filtrar nuestros resultados. Por ejemplo, si deseamos mostrar los sujetos de `censo` que tienen mas de dos hijos lo haríamos del siguiente modo:

```
subconjunto_censo <- censo[censo$nro_hijos > 2, ]
subconjunto_censo
##     nombre apellido fecha_nacimiento sexo nro_hijos
## 3 Ruben   Sancho     1958-12-25 HOMBRE        3
## 4 Daniel   Alfara     1983-09-19 HOMBRE        4
```

Añadir Filas y Columnas

Del mismo modo que utilizamos los operadores `[,]` y `$` para acceder y cambiar un valor, podemos añadir columnas en `censo` del siguiente modo:

```
# Añadimos la columna `nacionalidad` en `censo`
censo$nacionalidad <- c("ES", "FR", "RU", "IT")

# Mostramos `censo` `por pantalla
censo
##     nombre apellido fecha_nacimiento sexo nro_hijos nacionalidad
## 1 Juan    Sanchez     1976-06-14 HOMBRE        1       ES
## 2 Margarita Garcia      1974-05-07 MUJER        2       FR
## 3 Ruben   Sancho     1958-12-25 HOMBRE        3       RU
## 4 Daniel   Alfara     1983-09-19 HOMBRE        4       IT
```

Para añadir filas a un `dataframe` existente, definiremos un nuevo vector respetando las variables de las columnas que han sido definidas con anterioridad y pegando esta fila al `dataframe` original mediante la función `rbind()` (acrónimo de *rowbind*, pegar por filas):

```
# Definimos una nueva fila
fila_nueva <- c("Oscar", "Gonzalez", "1989-07-15", "HOMBRE", 0, "ES")

# Añadimos la nueva fila a `censo` con `rbind()`
censo <- rbind(censo, fila_nueva)

# Mostramos por pantalla `censo`
censo
##      nombre apellido fecha_nacimiento sexo nro_hijos
## 1      Juan Sanchez 1976-06-14 HOMBRE     1
## 2 Margarita Garcia 1974-05-07 MUJER     2
## 3    Ruben Sancho 1958-12-25 HOMBRE     3
## 4   Daniel Alfara 1983-09-19 HOMBRE     4
## 5   Oscar Gonzalez 1989-07-15 HOMBRE     0
```

Ordenación de DataFrames

Para ordenar un `data.frame` hemos de aplicar la función `order()` a la variable por la que queramos ordenar. Por ejemplo, si queremos ordenar el dataframe `censo` por orden alfabético de `nombre`, haríamos:

```
# Usamos `order()` para ordenar
censo[order(nombre), ]
##      nombre apellido fecha_nacimiento sexo nro_hijos
## 4   Daniel Alfara 1983-09-19 HOMBRE     4
## 1      Juan Sanchez 1976-06-14 HOMBRE     1
## 2 Margarita Garcia 1974-05-07 MUJER     2
## 3    Ruben Sancho 1958-12-25 HOMBRE     3
```

También podemos controlar la forma de ordenación mediante el argumento `decreasing`, el cual acepta los valores lógicos `TRUE` y `FALSE`. Por ejemplo, si deseamos ordenar los sujetos por el mayor número de hijos, lo haríamos:

```
censo[order(nro_hijos, decreasing = TRUE), ]
##      nombre apellido fecha_nacimiento sexo nro_hijos
## 4   Daniel Alfara 1983-09-19 HOMBRE     4
## 3    Ruben Sancho 1958-12-25 HOMBRE     3
## 2 Margarita Garcia 1974-05-07 MUJER     2
## 1      Juan Sanchez 1976-06-14 HOMBRE     1
```

Cambiar la Forma de un `data.frame` : formato `wide` y `long`

Cuando nos encontramos con múltiples valores, dispersos en varias columnas, para la misma instancia, nuestros datos están en forma `wide` (ancho).

Por otro lado, nuestros datos están en forma `long` (largo) si existe una observación (fila) por variable. Es decir, disponemos de múltiples filas por instancia.

Permitidme ilustrar estos conceptos mediante un ejemplo. Nuestros datos están en formato `long` cuando los colocamos del siguiente modo:

```
# Definimos las filas
alumno <- c(1, 2, 1, 2, 2, 1)
sexo <- c("M", "F", "M", "F", "F", "M")
asignatura <- c("Matematicas", "Ciencias", "Ciencias", "Literatura", "Matematicas",
               "Literatura")
nota <- c(10, 4, 8, 6, 7, 7)

# Creamos el `data.frame`
observaciones_formato_long <- data.frame(alumno, sexo, asignatura, nota)

# Mostramos por pantalla `formato_long`
observaciones_formato_long
##   alumno sexo  asignatura nota
## 1      1    M Matematicas  10
## 2      2    F   Ciencias   4
## 3      1    M   Ciencias   8
## 4      2    F Literatura   6
## 5      2    F Matematicas  7
## 6      1    M Literatura   7
```

Como podemos observar, existe una fila para cada variable que hemos definido. Este formato es muy útil en la gran mayoría de funciones estadísticas.

Por otro lado, nuestros datos se encuentran en formato `wide` cuando los organizamos del siguiente modo:

```
# Definimos las columnas
alumno <- c(1, 2)
sexo <- c("M", "F")
matematicas <- c(10, 7)
ciencias <- c(8, 4)
literatura <- c(7, 6)

# Creamos el `data.frame`
observaciones_formato_wide <- data.frame(alumno, sexo, matematicas, ciencias,
                                             literatura)

# Mostramos por pantalla `formato_wide`
observaciones_formato_wide
##   alumno sexo matematicas ciencias literatura
## 1      1     M          10        8         7
## 2      2     F           7        4         6
```

Como podemos observar cada columna (variable) representa un valor para cada instancia.

Uso de `stack()` Para Data Frames con Estructuras Simples

La función `stack()` básicamente combina varios vectores en un único vector. Siguiendo con el ejemplo, si queremos nuestro `data.frame` en formato `long` combinaremos las columnas `matematicas`, `ciencias` y `literatura` en una sola:

```
# Pasamos de `wide` a `long`
formato_long <- stack(observaciones_formato_wide, select = c(matematicas, ciencias,
                                                               literatura))

# Mostramos `formato_long`
formato_long
##   values      ind
## 1    10 matematicas
## 2     7 matematicas
## 3     8   ciencias
## 4     4   ciencias
## 5     7 literatura
## 6     6 literatura
```

Para pasar de formato `long` a `wide`, necesitamos desagrupar nuestros datos con el objetivo de obtener una fila por instancia, en la que cada valor se corresponde a una variable diferente. Para ello haremos uso de la función `unstack()` como mostramos en el siguiente ejemplo:

```
# Construimos a formato `wide`  
formato_wide <- unstack(observaciones_formato_long, nota ~ asignatura)  
  
# Devuelve `formato_wide`  
formato_wide  
## Ciencias Literatura Matematicas  
## 1      4      6      10  
## 2      8      7      7
```

Transformar Data Frames con Estructuras Complejas

Existen otras funciones que nos permiten transformar data frames con estructuras más complejas:

- La función `reshape()` que forma parte del paquete `stats`.
- Las funciones `spread()` y `gather` del paquete `tidyverse`.
- La función `melt()` del paquete `reshape2`.

En este curso en un [módulo posterior](#) profundizaremos en el uso del paquete `tidyverse`.

Resumen

- Creamos data frames con `data.frame()`.
- Mediante la función `str()` lograremos conocer la estructura del data frame.
- Para inspeccionar las primeras o últimas líneas, podemos hacer uso de las funciones `head()` y `tail()`.
- Por medio de `nrow()` y `ncol()` recuperamos el número de filas y columnas.
- Accederemos a los elementos de una data frame del mismo modo que lo hacemos en las matrices o listas.
- Gracias a las funciones `attach` y `detach()` podemos acceder a las variables de un data frame de forma directa.
- Hemos aprendido a eliminar, añadir y ordenar filas y columnas.
- Por último, hemos discutido cuando nos encontramos en formato `wide` y `long`.

Estructuras de Control

También llamadas como, **estructura básica** o **sentencia de control** se utilizan para controlar el flujo de un programa (o bloque de instrucciones), son métodos que permiten especificar el orden en el cual se ejecutarán las instrucciones en un algoritmo. Si no existieran las **estructuras de control**, los programas se ejecutarían linealmente desde el principio hasta el fin, sin la posibilidad de tomar decisiones.

Por lo general, en la mayoría de lenguajes de programación encontraremos dos tipos de estructuras de control. Encontraremos un tipo que permite la ejecución condicional de bloques de código y que son conocidas como **estructuras condicionales**. Por otro lado, encontraremos las **estructuras iterativas** que permiten la repetición de un bloque de instrucciones, un número determinado de veces o mientras se cumpla una condición.

De acuerdo con el manual base de R, las estructuras condicionales corresponden a las palabras reservadas `if` y su variante con `else`. Por otro parte, las estructuras para bucles son `for`, `while` y `repeat`, con las cláusulas adicionales `break` y `next`. Podemos consultar las diferentes estructuras de control mediante el comando `?Control` en RStudio.

Objetivos

Después de leer este capítulo, deberíamos:

- Ser capaces de controlar el flujo de ejecución con la ayuda de las estructuras condicionales.
- Estar capacitado para ejecutar bloques de código repetidamente mediante estructuras iterativas.

Estructuras Condicionales

Las estructuras condicionales permiten la ejecución condicional de bloques de código. La instrucción `if`, que en inglés tiene el mismo significado que nuestro "si" condicional, permite evaluar una expresión y, sobre la base de su resultado (verdadero o falso), ejecutar o no la instrucción o el bloque que le sigue. Es decir, si el resultado de la expresión es verdadero, ejecuta el código. En caso contrario, el programa sigue su curso normal.

A continuación, mostramos el [diagrama](#) de flujo de `if`. En términos de diagramas de flujo, los rectángulos significan la realización de un proceso, en otras palabras la ejecución de un bloque de instrucciones. Por otro lado, los rombos son conocidos como símbolos de decisión, es decir se corresponden a preguntas cuyas respuestas únicamente tienen dos posibles respuestas, concretamente, TRUE (T) o FALSE (F).

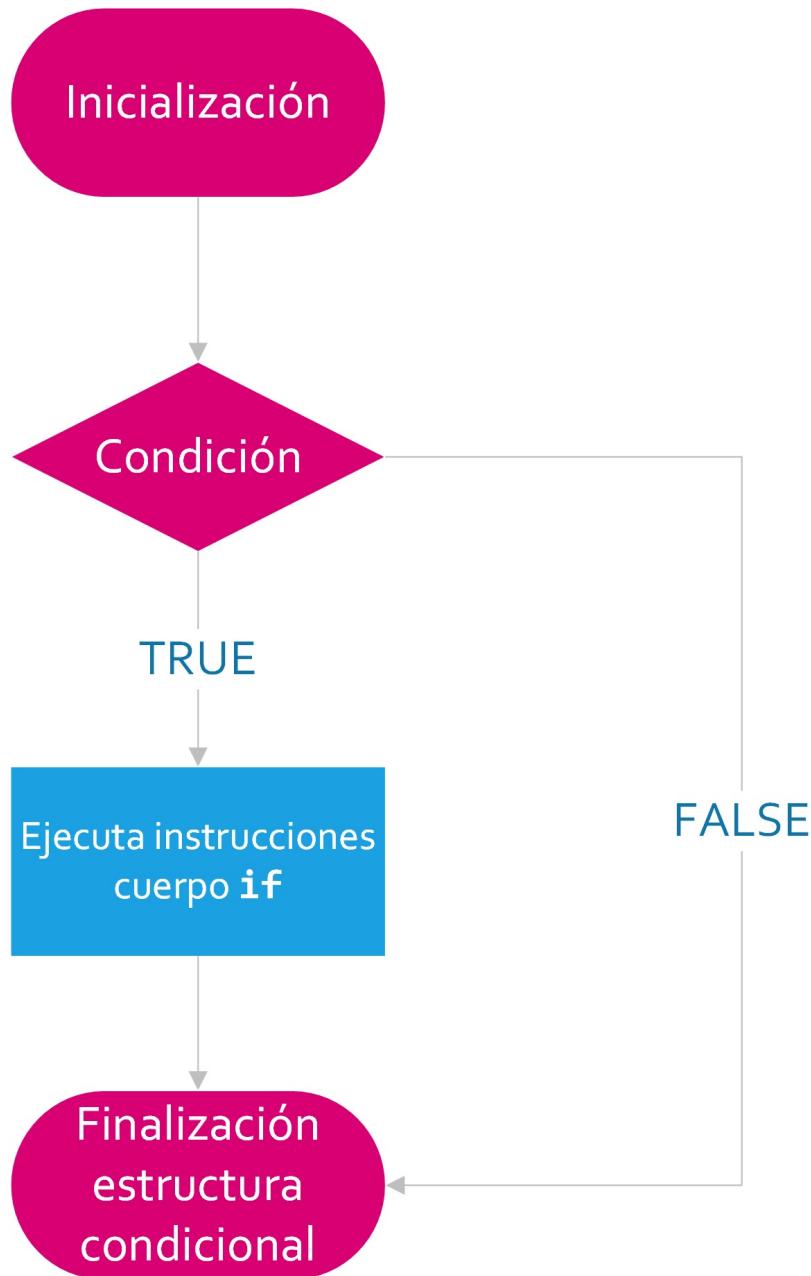


Figura1 - Diagrama de Flujo If

La instrucción `if` toma un valor lógico (en realidad, un vector lógico de longitud uno) y ejecuta la siguiente instrucción sólo si el valor es `TRUE` :

```
if (TRUE) {  
  message("Es verdadero, se ejecutara la instrucion.")  
}  
## Es verdadero, se ejecutara la instrucion.
```

```
if (FALSE) {
  message("Es falso, no se ejecutara la instrucion.")
}
```

En el caso que pasemos valores desconocidos (NA) a `if` , R lanzará un error:

```
if (NA) {
  message("Los valores desconocidos lanzan un error")
}
## Error in if (NA) {}: missing value where TRUE/FALSE needed
```

Si nos encontramos ante esta situación, deberíamos comprobarlo mediante la función

`is.na()` :

```
if (is.na(NA)) {
  message("El valor es desconocido.")
}
## El valor es desconocido.
```

Desde luego, en nuestro código en pocas ocasiones pasaremos los valores `TRUE` y `FALSE` a la instrucción `if` . En su lugar, pasaremos una variable o expresión. En el siguiente ejemplo, `runif(1)` genera un número de forma aleatoria entre 0 y 1. Si el valor es mayor que `0.5` , entonces el mensaje será mostrado:

```
if (runif(1) > 0.5) {
  message("Este mensaje aparece con un 50% de probabilidad.")
```

Si pretendemos ejecutar un bloque de instrucciones, podemos envolverlo entre paréntesis:

```
x <- 3
if (x < 2) {
  y <- 2 * x
  z <- 3 * y
}
```

Recuerda que para que nuestro código sea lo mas legible posible, algunas [guías de estilo](#) recomiendan el uso de paréntesis, incluso si sólo queremos ejecutar condicionalmente una sentencia.

El siguiente paso en complejidad en la sintaxis de `if` es incluir la cláusula `else` . Seguidamente se muestra el [diagrama de flujo](#) de `if-else` :

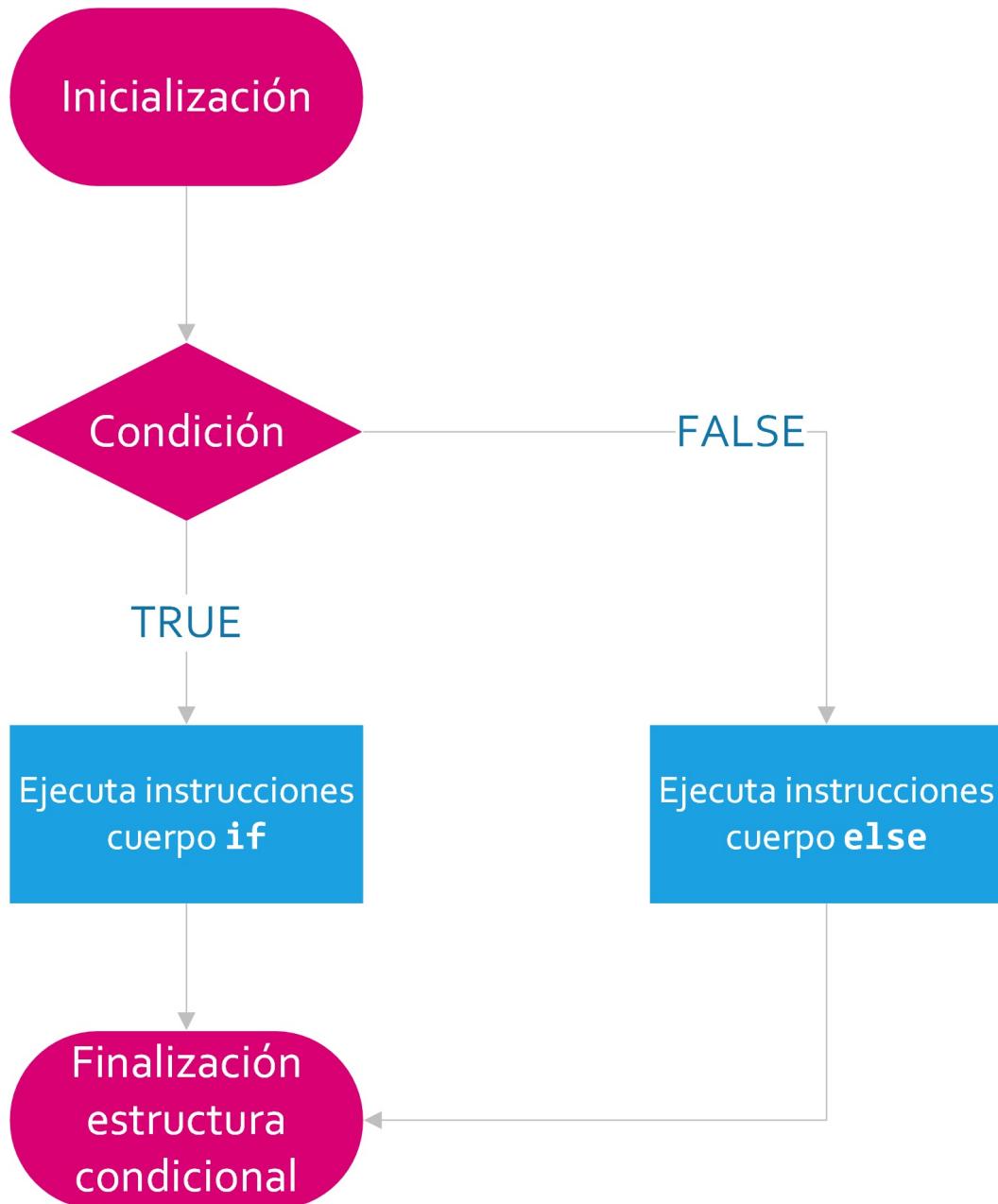


Figura2 - Diagrama de Flujo if-else

El código después de un `else` se ejecuta sólo si la condición en `if` es `FALSE` :

```

if (FALSE) {
  message("Este bloque no se ejecuta...")
} else {
  message("pero este si lo hará")
}
## pero este si lo hará
  
```

Podemos definir múltiples condiciones combinando `if` y `else` repetidamente, este tipo de estructura se utiliza para probar **condiciones mutuamente excluyentes**.

A continuación se muestra el [diagrama de flujo](#) de un `if-else` anidado. Como podemos observar se pueden plantear múltiples condiciones simultáneamente: si se cumple la condición 1 se ejecuta el bloque de instrucciones 1. En caso contrario se comprueba la condición 2; si es cierta se ejecuta el bloque de sentencias 2, y así sucesivamente hasta `n` condiciones. Si ninguna de ellas se cumple se ejecuta el bloque de instrucciones de `else`:

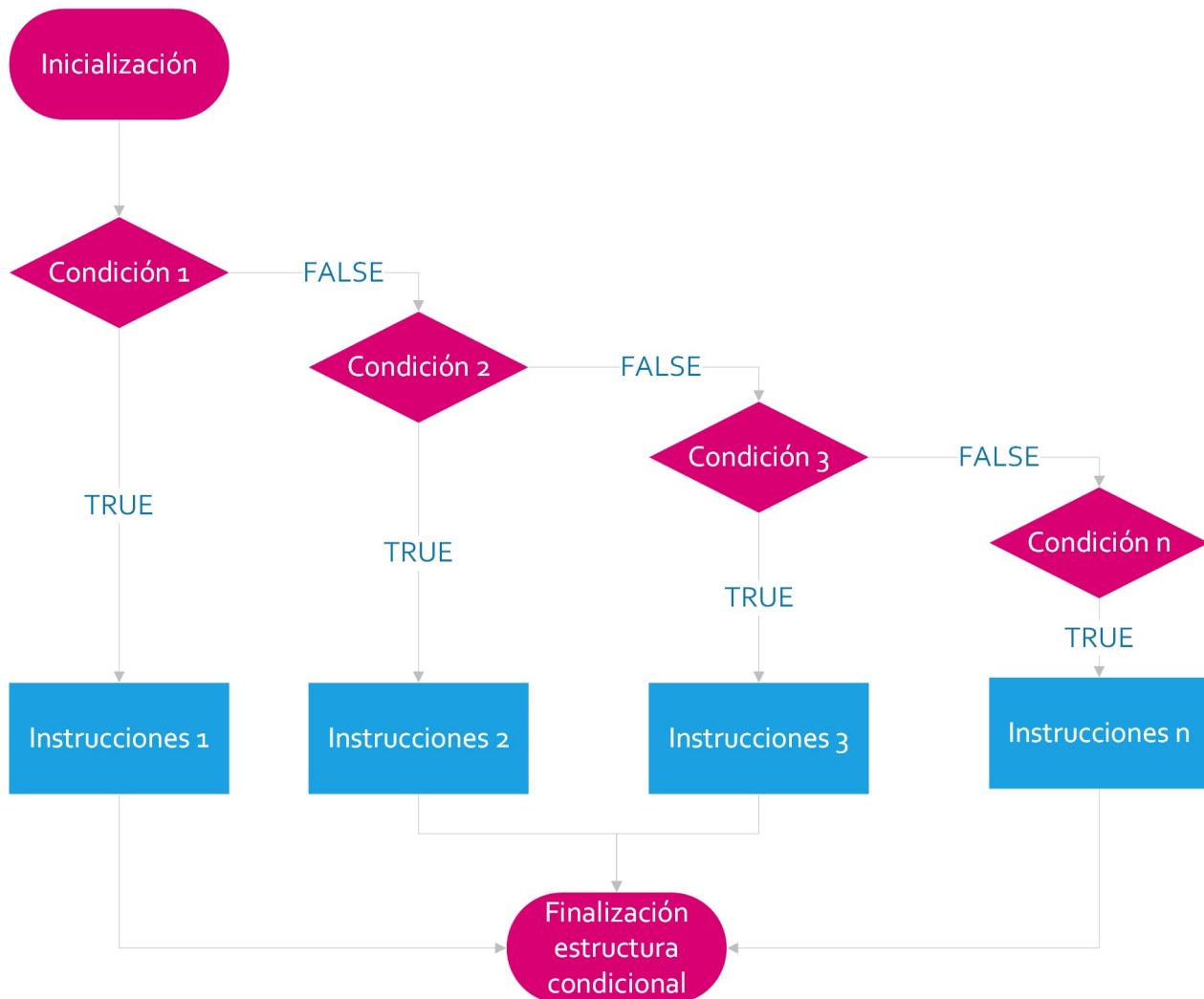


Figura3 - Diagrama de Flujo If-Else Anidados

El siguiente ejemplo nos sirve para mostrar el anidamiento de instrucciones `if-else`. Pongamos el caso que deseamos crear un algoritmo que nos calcule la [medida de tendencia central](#) que deseemos, el siguiente fragmento de código podría ser una posible solución:

```
## Error in library(modeest): there is no package called 'modeest'
```

```
# Creamos una muestra de 20 observaciones del 1 al 100 en el que se pueden
# repetir hasta 2 observaciones
(muestra <- sample(1:100, 20, 2))
## [1] 66 18 32 26 75 23 10 46 15 22 5 42 61 71 71 5 61 73 39 35

## Creamos una variable indicando la medida de tendencia central que queremos
## calcular
centralizacion <- "moda"

## Creamos un algoritmo para calcular la tendencia central que deseemos
if (centralizacion == "moda") {
  media = mean(muestra)
  message("La media es ", as.character(media))
} else if (centralizacion == "mediana") {
  mediana = median(muestra)
  message("La mediana es ", as.character(mediana))
} else if (centralizacion == "moda") {
  moda = mlv(muestra, method = "mfv")
  message("La moda es ", as.character(moda))
} else {
  message("Este algoritmo sola calcula la media,
          mediana, moda")
}
## La media es 39.8
```

If Vectorizado

La instrucción `if` estándar acepta en la condición lógica un único valor lógico. Si pasamos un vector lógico con una longitud mayor que uno, R nos lo indicará mediante un **warning** indicándonos que hemos introducido múltiples opciones, pero que únicamente la primera será utilizada:

```
if (c(TRUE, FALSE)) {
  message("dos condiciones")
}
## Warning in if (c(TRUE, FALSE)) {: the condition has length > 1 and only the
## first element will be used
## dos condiciones
```

Puesto que muchas de las operaciones en R son vectorizadas, R nos proporciona la función `ifelse`. La función `ifelse` toma tres argumentos. El primer argumento es un vector lógico de condiciones. El segundo es un vector que contiene los valores que serán devueltos

cuando el primer vector es `TRUE`. El tercero contiene los valores que serán devueltos cuando el primer vector es `FALSE`.

Uso

```
str(ifelse)
## function (test, yes, no)
```

Argumentos

```
test: un objeto que pueda ser coercionado a un tipo lógico
yes: devuelve los valores `TRUE` en los elementos de `test`
no: devuelve los valores `FALSE` de `test`
```

En el siguiente ejemplo, la función `rbinom` genera números aleatorios de un distribución binomial simulando el lanzamiento de una moneda:

```
ifelse(rbinom(n = 10, size = 1, prob = 0.5), "cara", "cruz")
## [1] "cara" "cara" "cara" "cara" "cruz" "cara" "cara" "cruz" "cara" "cara"
```

No obstante, `if(test) yes else no` es mucho mas eficiente y preferible a `ifelse(test, yes, no)` cuando `test` es decir, la condición lógica se trata de una expresión cuya longitud sea un vector de longitud igual a 1.

Selección Múltiple con `switch()`

El código con muchas cláusulas `else` puede hacer nuestro código difícil de leer. En estas circunstancias, podemos hacer uso de la función `switch()`, con la que conseguiremos un código mas legible y fácil de mantener.

Para comprender mejor su funcionamiento pasemos a examinar el [diagrama de flujo](#) de la función `switch()`:

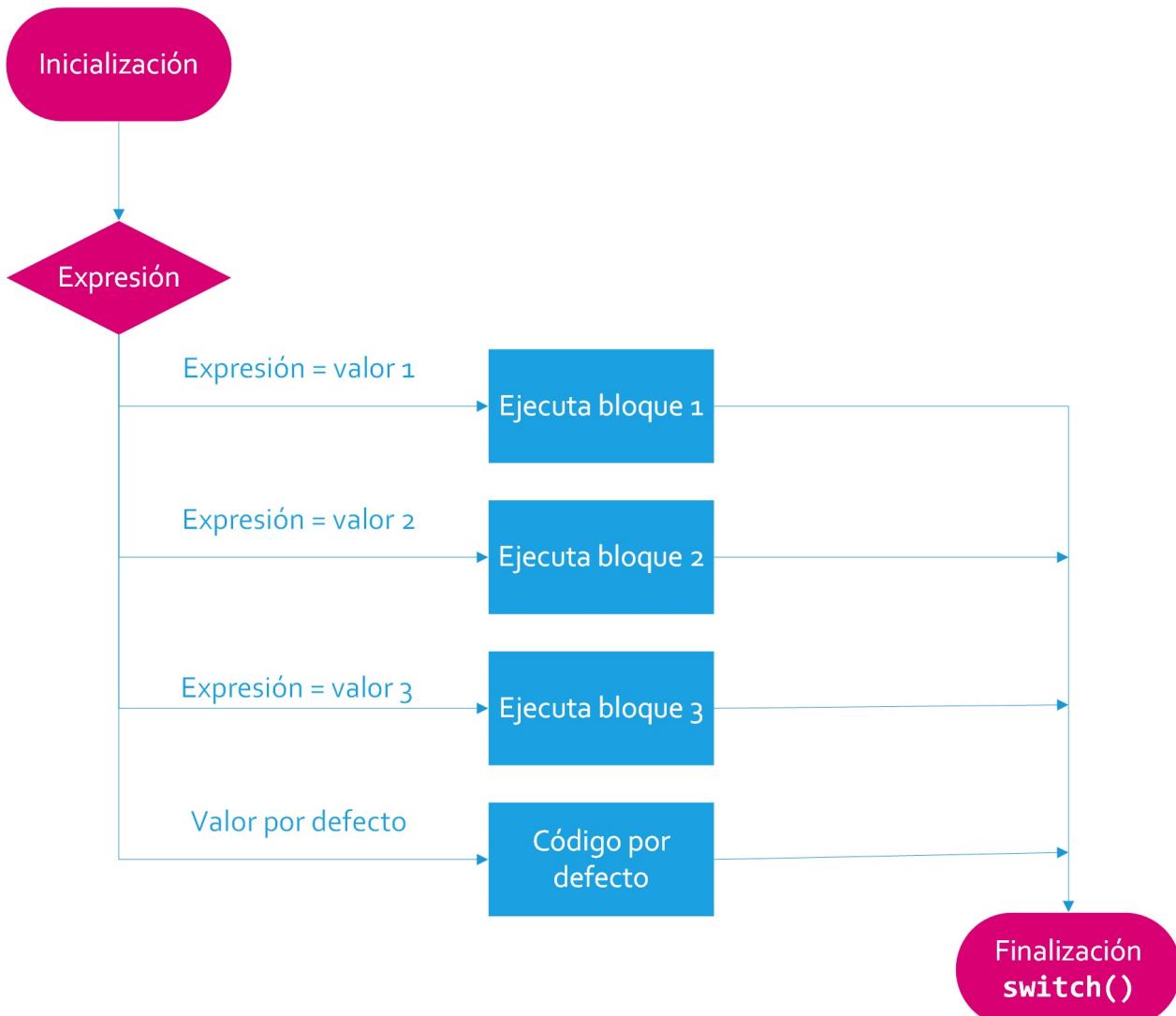


Figura4 - Diagrama de Flujo switch()

Esta función permite ejecutar una de entre varias acciones en función del valor de una expresión. Es una alternativa a los `if-else` anidados cuando se compara la misma expresión con diferentes valores.

El caso más común toma como primer argumento una expresión que devuelve un string, seguido por varios argumentos con nombre que proporcionan el resultado cuando el nombre coincide con el primer argumento. Cuando se encuentra la primera coincidencia, se ejecuta el bloque de instrucciones y se sale de función. Si no se encuentra ninguna coincidencia con ningún valor, se ejecuta el bloque de sentencias del argumento por defecto.

Uso

```
str(switch)
## function (EXPR, ...)
```

Argumentos

EXPR: una expresión que evalua un string
... : una lista con alternativas. Estas alternativas se les dará nombre, excepto a aquella que sea usada como valor por defecto.

Una alternativa al ejemplo presentado en el apartado anterior mediante la función `switch()` es la siguiente:

```
## Error in library(modeest): there is no package called 'modeest'
```

```
# Creamos una muestra de 20 observaciones del 1 al 100 en el que se pueden
# repetir hasta 2 observaciones
(muestra <- sample(1:100, 20, 2))
## [1] 53 93 42 55 33 6 92 14 97 53 63 59 43 97 69 51 29 61 72 55

# Calculamos la media de la muestra
(switch("media", media = mean(muestra), mediana = median(muestra), moda = mlv(muestra,
method = "mfv")))
## [1] 56.85
```

Si ningún nombre coincide, entonces `switch` devuelve `NULL`:

```
# Intentamos calcular la desviación típica
(switch("desviacion_tipica", media = mean(x), mediana = median(x), moda = mlv(x,
method = "mfv")))
## NULL
```

En este escenario, podemos proporcionar un argumento por defecto sin nombre que `switch()` devolverá cuando no coincida ningún otro:

```
# Intentamos calcular la desviación típica
(switch("desviacion_tipica", media = mean(x), mediana = median(x), moda = mlv(x,
method = "mfv"), "Solo se puede calcular la media, mediana y moda"))
## [1] "Solo se puede calcular la media, mediana y moda"
```


Estructuras Iterativas

Las instrucciones de repetición, de iteración o bucles, facilitan la repetición de un bloque de instrucciones, un número determinado de veces o mientras se cumpla una condición.

Por lo general, existen dos tipos de estructuras iterativas o bucles en los lenguajes de programación. Encontraremos un tipo de bucle que se ejecuta un número preestablecido de veces, que es controlado por un contador o índice, incrementado en cada iteración. Este tipo de bucle forma parte de la familia `for`.

Por otro lado, encontraremos un tipo de bucle que se ejecuta mientras se cumple una condición. Esta condición se comprueba al principio o el final de la construcción. Esta variante pertenece a la familia `while` or `repeat`, respectivamente.

Por último, siempre podemos consultar los comandos de control del flujo mediante `? control` en la consola de RStudio.

Bucle `for`

El bucle `for` es una estructura iterativa que se ejecuta un número preestablecido de veces, que es controlado por un contador o índice, incrementado en cada iteración.

A continuación, mostramos el diagrama de flujo del bucle `for`. En términos de diagramas de flujo, los rectángulos significan la realización de un proceso, en otras palabras la ejecución de un bloque de instrucciones. Por otro lado, los rombos con conocidos como símbolos de decisión, es decir se corresponden a preguntas cuyas respuestas únicamente tienen dos posibles respuestas, concretamente, TRUE (T) o FALSE (F).

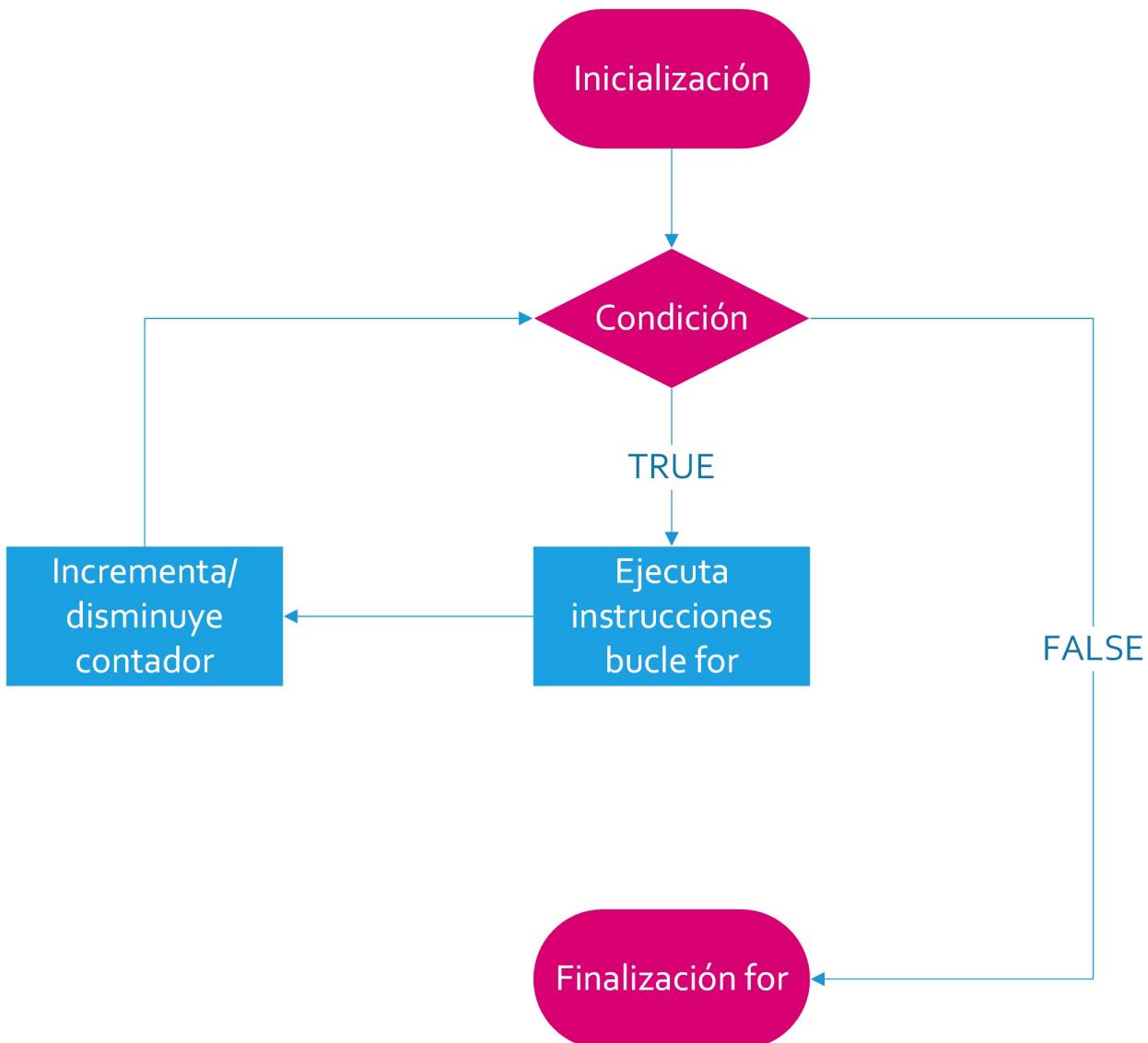


Figura1 - Diagrama de Flujo Bucle For

Una o mas instrucciones dentro del rectángulo de inicialización son seguidas por la evaluación de la condición en una variable la cual puede asumir valores dentro de una secuencia. En la [figura](#), esto es representado por el símbolo del rombo.

En otras palabras, estamos comprobando si el valor actual de la variable está dentro de un rango específico. Por lo general, especificaremos el rango en la inicialización.

Si la condición no se cumple, es decir el resultado es `False`, el bucle nunca se ejecutará. Esto es indicado por la flecha de la derecha de la estructura `for`. El programa entonces ejecutará la primera instrucción que se encuentre después del bucle.

Si la condición se verifica, una instrucción o bloque de instrucciones es ejecutado. Una vez la ejecución de estas instrucciones ha finalizado, la condición es evaluada de nuevo. En la [figura](#) esto es indicado por las líneas que van desde el rectángulo que incrementa o

disminuye el contador hasta el símbolo del rombo que evalúa la condición.

Por ejemplo, en el siguiente fragmento de código calculamos la media de un conjunto de observaciones, que se obtiene dividiendo la suma de todas las observaciones por el número de individuos:

```
# Creamos un vector aleatorio de 10 observaciones
observaciones <- sample(1:50, 100, replace = TRUE)

# Inicializamos `suma` de todas las observaciones
suma <- 0

# Creamos un bucle for que calcula la media
for (i in seq_along(observaciones)) {
  suma <- observaciones[i] + suma
  media <- suma/length(observaciones)
}

# Mostramos por pantalla la media
media
## [1] 25.99
```

Bucles `for` Anidados

Los bucles `for` pueden ser anidados. En el siguiente fragmento de código creamos un algoritmo que calcula la suma de dos matrices cuadradas:

```
# Creamos dos matrices cuadradas
m1 <- matrix(sample(1:100, 9, replace = TRUE), nrow = 3)
m2 <- matrix(sample(1:100, 9, replace = TRUE), nrow = 3)

# Inicializamos la matriz que contendrá m1+m2
suma <- matrix(nrow = 3, ncol = 3)

# Para cada fila y cada columna, realizamos la suma elemento a elemento
for (i in 1:nrow(m1)) {
  for (j in 1:ncol(m1)) {
    suma[i, j] <- m1[i, j] + m2[i, j]
  }
}

# Mostramos por pantalla la suma de m1+m2
suma
##      [,1] [,2] [,3]
## [1,] 113 162  42
## [2,] 143  90 146
## [3,] 119 109 110
```

El siguiente ejemplo sirve para exemplificar el anidamiento de bucles `for`. Cada uno con su propio bloque de instrucciones y manejado con su propio índice. Es decir, `i` controla las filas de las matrices y `j` las columnas.

Bucle `while`

Cuando nos encontramos en la situación en la que no conocemos el número de iteraciones de antemano, podemos hacer uso del bucle `while`. Este bucle se ejecuta mientras se cumple una condición que se comprueba al principio de la construcción.

A continuación se muestra el diagrama de flujo de `while`:

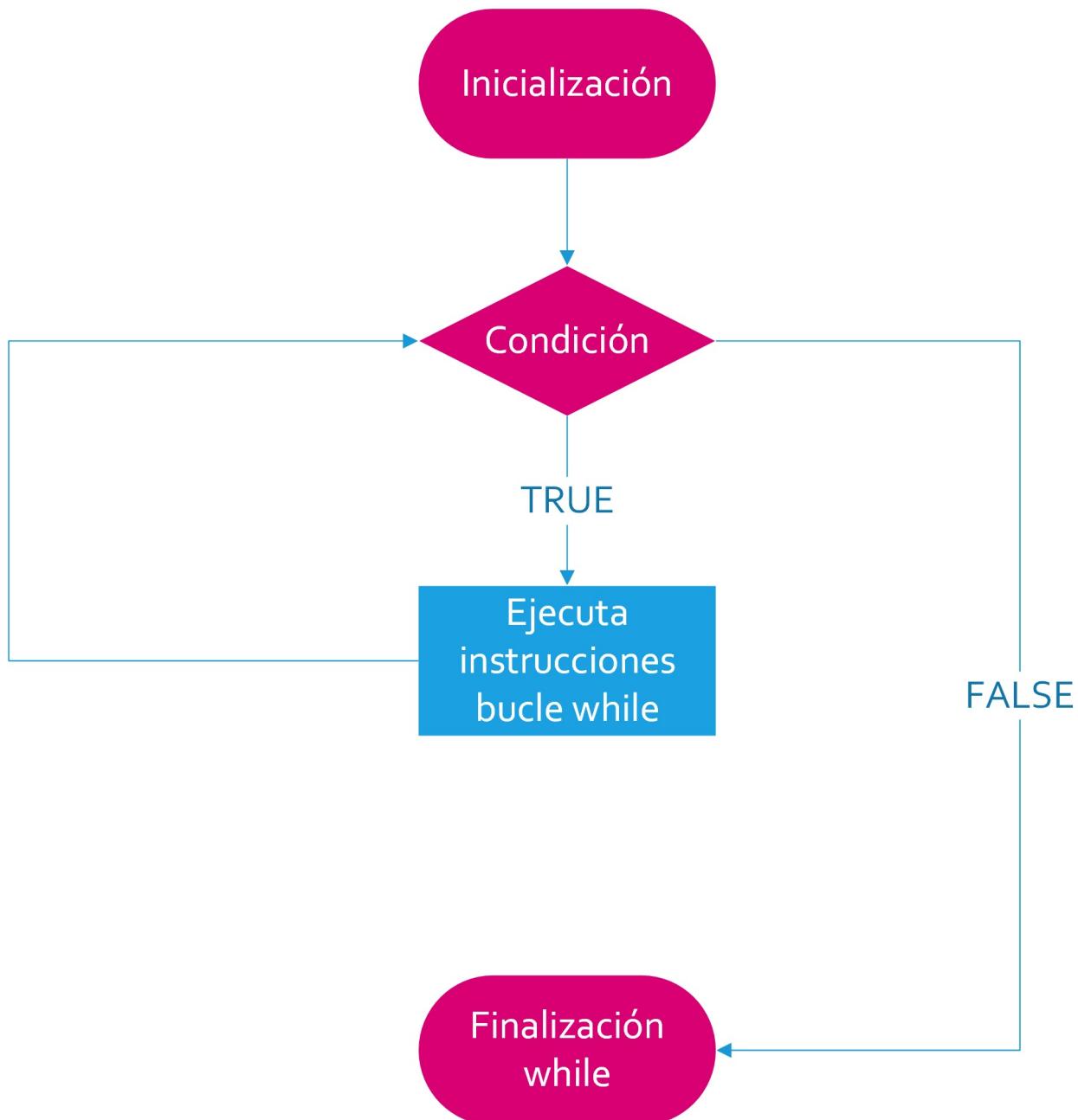


Figura2 - Diagrama de Flujo Bucle While

La estructura de una construcción `while` está compuesta de un bloque de inicialización, seguido por una condición lógica. Esta condición es normalmente una expresión de comparación entre una variable de control y un valor, en la que usaremos los operadores de comparación, pero cabe señalar que cualquier expresión que evalúa a un valor lógico, `TRUE` o `FALSE`, es válida.

Si el resultado es `FALSE` (F), el bucle nunca será ejecutado. Esto es indicado por la flecha de la derecha en la figura. En esta situación el programa ejecutará la primera instrucción que encuentre después del bloque iterativo.

Por otro lado, si el resultado es `TRUE` (T), la instrucción o bloque de instrucciones del cuerpo de `while` son ejecutadas. Esto sucederá hasta que la condición lógica sea `FALSE`.

El siguiente ejemplo es un ejemplo de utilización de la estructura `while`:

```
# Algoritmo que muestra por pantalla los 10 primeros números naturales
n = 1
while (n <= 5) {
  print(n)
  n = n + 1
}
## [1] 1
## [1] 2
## [1] 3
## [1] 4
## [1] 5
```

Bucle `repeat`

El bucle `repeat` es similar a `while`, excepto que la instrucción o bloque de instrucciones de `repeat` es ejecutado al menos una vez, sin importar cual es el resultado de la condición.

A continuación, como en los apartados anteriores mostramos el [diagrama de flujo](#) de la estructura `repeat`:

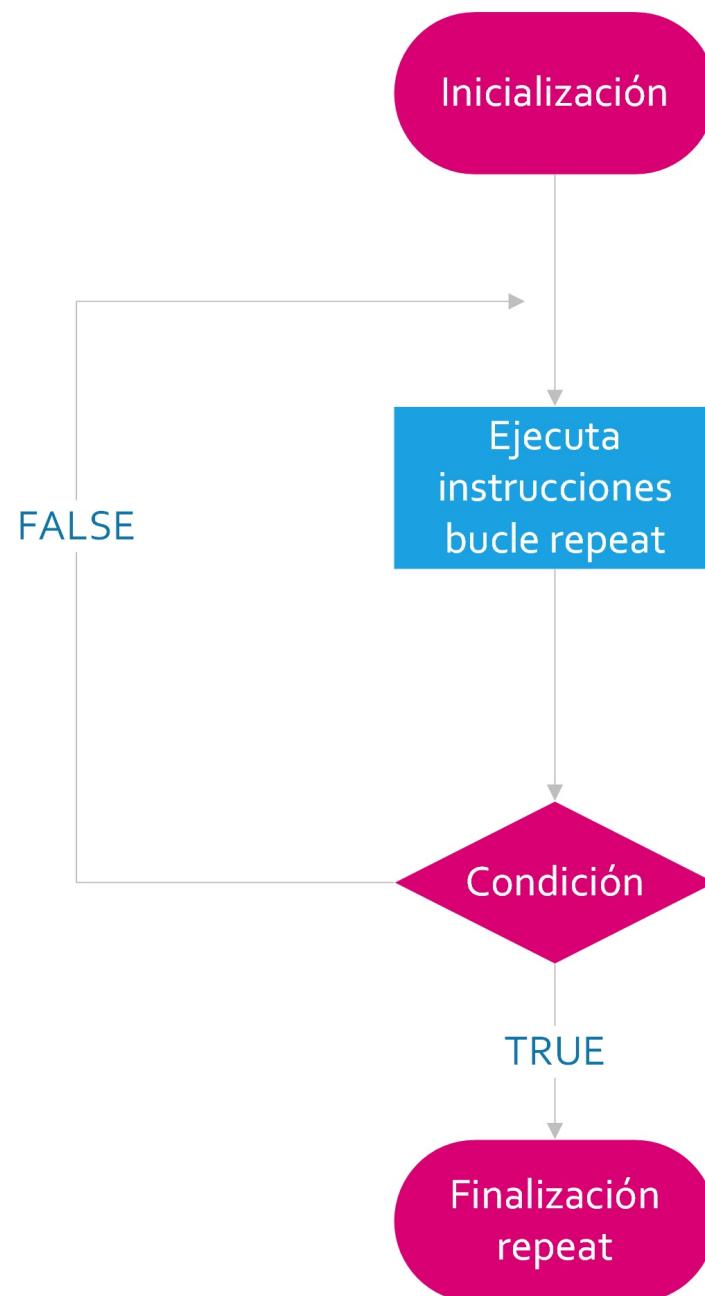


Figura3 - Diagrama de Flujo Repeat

Como alternativa al ejemplo anterior, podríamos codificar el algoritmo como:

```
# Algoritmo que muestra por pantalla los 10 primeros números naturales
n = 1
repeat {
    if (n <= 10) {
        print(n)
        n = n + 1
    } else {
        break
    }
}
## [1] 1
## [1] 2
## [1] 3
## [1] 4
## [1] 5
## [1] 6
## [1] 7
## [1] 8
## [1] 9
## [1] 10
```

En el ejemplo de la estructura `repeat` podemos observar que el bloque de código es ejecutado al menos una vez y que finaliza cuando la función `if` es verificada.

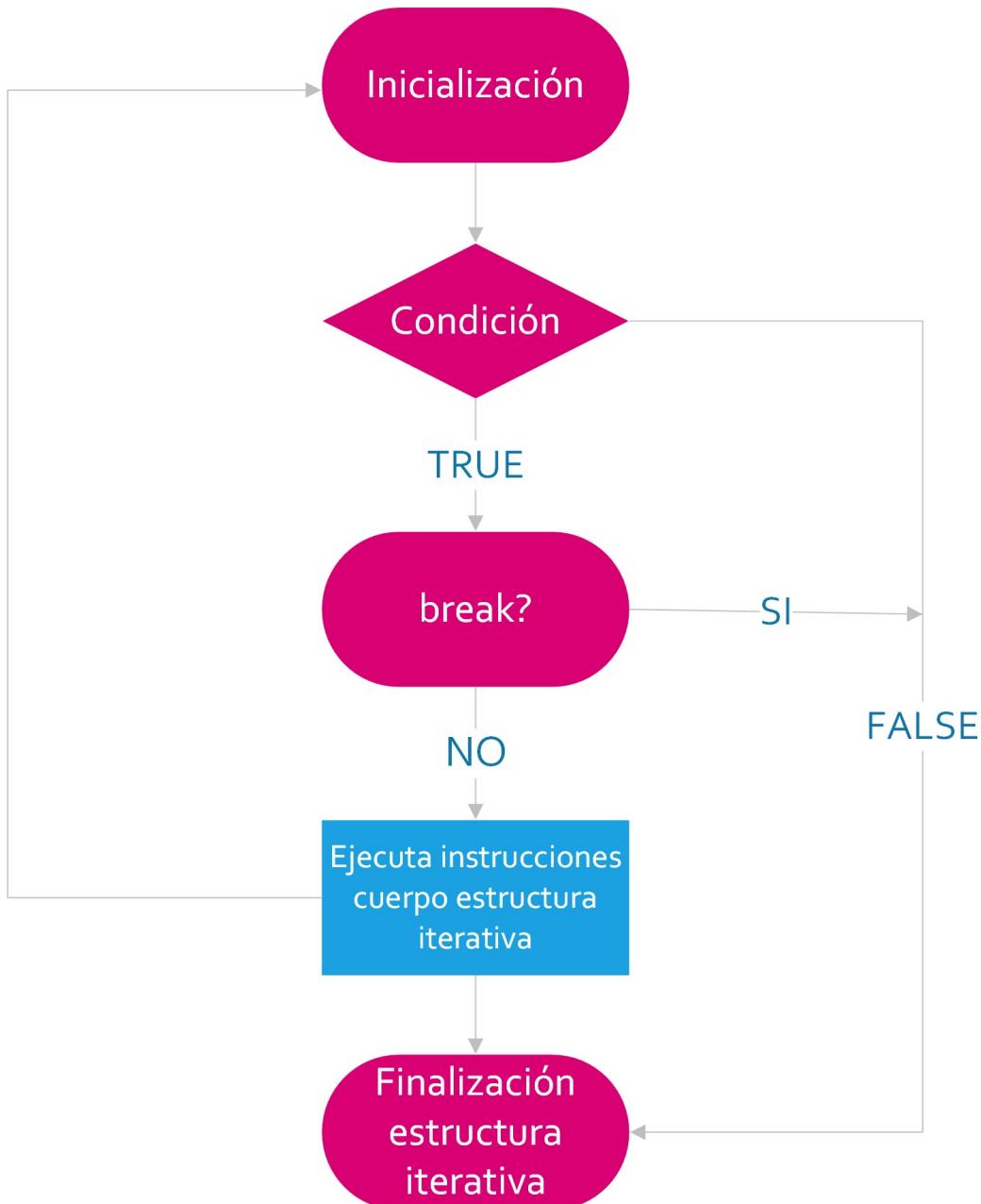
Observemos que hemos tenido que establecer una condición dentro del bucle la cual establece la salida con la cláusula `break`. Esta cláusula nos introduce en el concepto de **salida de las iteraciones** en los bucles y que pasamos a analizar en el apartado siguiente.

Cláusula `break`

La instrucción `break` se utiliza con las instrucciones de bucle `for`, `while` y `repeat`.

La cláusula `break` finaliza la ejecución del bucle más próximo en el que aparece. El control pasa a la instrucción que hay a continuación del final de la instrucción, si hay alguna.

A continuación se muestra el [diagrama de flujo](#) de `break`:

*Figura4 - Diagrama Flujo Break*

Como podemos ver en el diagrama de flujo, la instrucción `break` finaliza la ejecución de la instrucción envolvente `for`, `while` o `repeat` mas próxima. El control pasa a la instrucción que hay a continuación de la instrucción finalizada, si hay alguna.

Para ilustrar mejor el uso de `break` crearemos un algoritmo que define una **matriz cuadrada** y que utiliza dos bucles `for` anidados para calcular la **diagonal principal** y su **matriz triangular superior**:

```

# Creamos una matriz cuadrada de 6 x 6
m <- matrix(data = sample(x = 10, size = 36, replace = TRUE), nrow = 6, ncol = 6)
# Mostramos por pantalla `m`
m
##      [,1] [,2] [,3] [,4] [,5] [,6]
## [1,]    10    1    2    5    8    6
## [2,]     8    8    3    9    8    7
## [3,]     8    9    2    5    1    7
## [4,]     4    3    3    4    4    1
## [5,]     6    5    8    8    7    7
## [6,]    10    3    1    2    2    3

# Creamos un vector para la diagonal principal
diagonal_principal <- vector(mode = "integer", length = nrow(m))
diagonal_principal
## [1] 0 0 0 0 0 0

# Algoritmo que calcula la matriz triangular inferior y su diagonal
# principal
for (i in 1:nrow(m)) {
  for (j in 1:ncol(m)) {
    if (i == j) {
      break
    } else {
      m[i, j] <- 0
    }
  }
  diagonal_principal[j] <- m[i, j]
}

# Mostramos por pantalla diagonal principal
diagonal_principal
## [1] 10 8 2 4 7 3
# Mostraamos por pantalla matriz inferior de m
m
##      [,1] [,2] [,3] [,4] [,5] [,6]
## [1,]    10    1    2    5    8    6
## [2,]     0    8    3    9    8    7
## [3,]     0    0    2    5    1    7
## [4,]     0    0    0    4    4    1
## [5,]     0    0    0    0    7    7
## [6,]     0    0    0    0    0    3

```

Examinaremos brevemente ahora el código anterior, como se puede observar en primer lugar se define una matriz cuadrada de 6 x 6 y creamos un vector de tipo entero con una longitud de 6 que en el momento de su inicialización contiene todos sus valores igual a cero.

Cuando los indices son iguales cumpliéndose la condición del bucle `for` mas interno, y que itera mediante `j` por las columnas de la matriz, se ejecuta un `break` y el loop mas interno es interrumpido para saltar directamente a la instrucción del bucle mas externo. Esta instrucción calcula la diagonal principal de la matriz. Seguidamente, el control pasa la condición lógica del bucle más externo que itera por las filas de `m` mediante el índice `i`.

En el caso de que los indices sean diferentes, a la posición del elemento `[i, j]` se le asigna el valor de cero con el propósito de calcular la matriz triangula superior.

Por otro lado, dentro de instrucciones anidadas, la instrucción `break` finaliza solo la instrucción `for`, `while` o `repeat` que la envuelve inmediatamente. Podemos utilizar la instrucción `next` para transferir el control desde estructuras más anidadas. Esta cláusula nos introduce en el concepto de **interrupción de las iteraciones** en una estructura iterativa y que pasamos a analizar en el apartado siguiente.

Cláusula `next`

La cláusula `next` interrumpe una iteración y salta al siguiente ciclo. De hecho, salta a la evaluación de la condición del bucle en el que se encuentra.

En otros lenguajes de programación el equivalente a `next` es conocido como **continue**, cuyo significado es el mismo: en la línea de código que te encuentres, bajo la verificación de la condición, salta a la evaluación del bucle.

El diagrama de flujo de `break` se muestra en la [figura](#) siguiente:

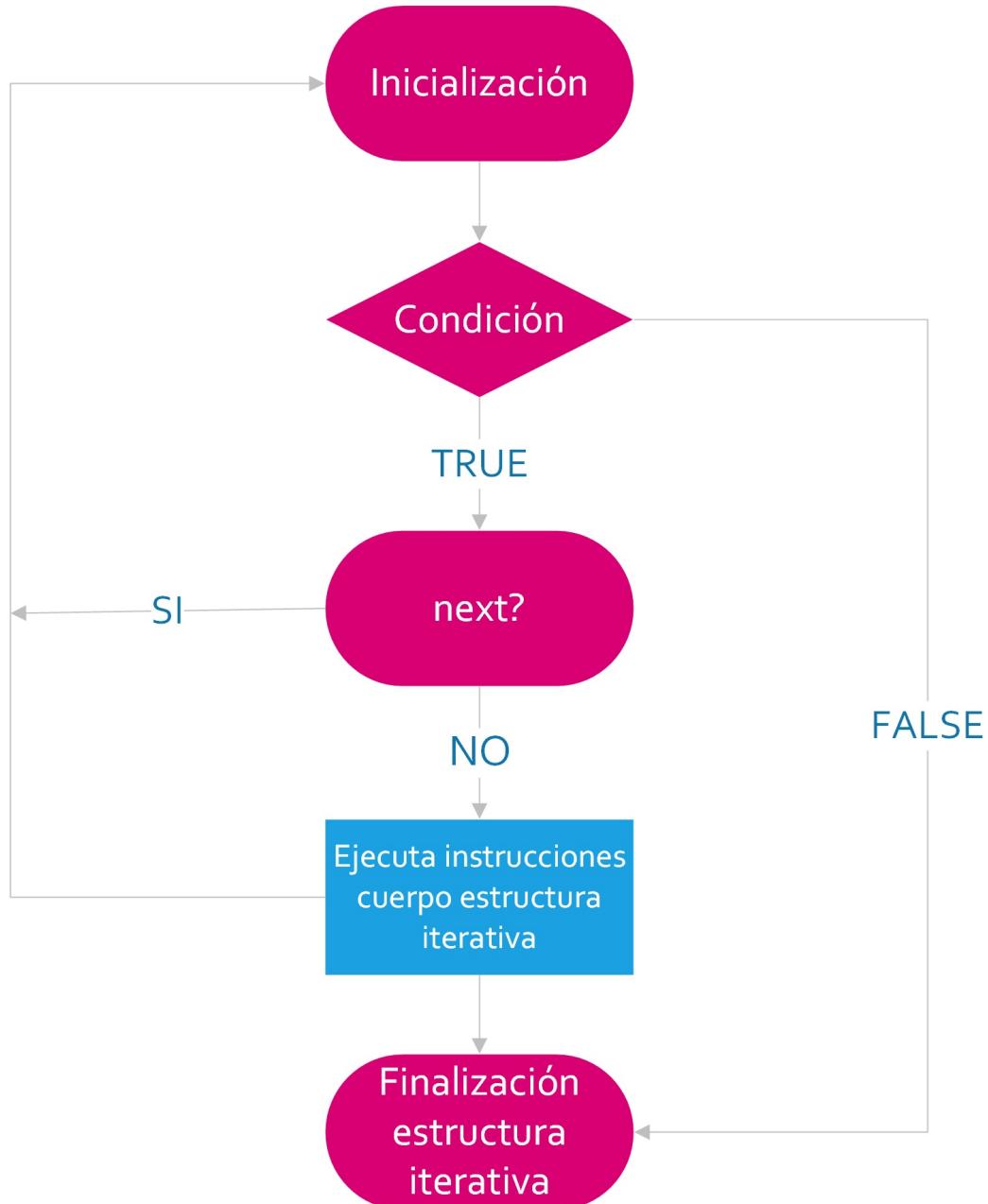


Figura5 - Diagrama flujo Next

Examinemos a continuación el diagrama de flujo de la instrucción `break`. Como hemos comentado con anterioridad `break` fuerza la transferencia del control a la expresión de control del bucle contenedor `for`, `while` o `repeat` más pequeño. Es decir, no se ejecuta ninguna de las instrucciones restantes de la iteración actual. La siguiente iteración del bucle se determina del modo siguiente:

- En un bucle `while` o `repeat`, la siguiente iteración se inicia reevaluando la expresión de control de la instrucción `while` o `repeat`.

- En un bucle `for`, en primer lugar se incrementa el índice del bucle. A continuación, se evalúa de nuevo la expresión de control de la instrucción `for` y, en función del resultado, el bucle finaliza o se produce otra iteración.

Pongamos por caso que queremos mostrar por pantalla los números pares de una secuencia de enteros:

```
for (i in 1:10) {
  if (i%%2)
    next
  print(i)
}
## [1] 2
## [1] 4
## [1] 6
## [1] 8
## [1] 10
```

Este algoritmo utiliza el [teorema del resto](#) para calcular si un número es par o impar. Si el resto de dividir el número entre dos es igual a cero entonces se trata de un número par y es mostrado por pantalla.

Por otro lado, si el resto es diferente a cero será impar y saltará a la evaluación de la condición del bucle `for` ignorando cualquier instrucción que se encuentre a continuación.

Hay que mencionar, además el uso del operador `%%` para calcular el resto de la división especificado en la condición lógica de la instrucción `if`.

Alternativas al Uso de Bucles en R

Hasta el momento hemos visto el uso de bucles en R, pero podemos estar preguntándonos, ¿Cuando usar los bucles en R y cuando no?

Como regla general, si nos encontramos en la situación que tenemos que llevar a cabo una tarea que requiere tres o mas repeticiones, entonces una instrucción iterativa puede ser útil. Esto hace que nuestro código sea mas compacto, legible y mas fácil de mantener.

Sin embargo, la peculiar naturaleza de R sugiere no hacer uso de los bucles en todas las situaciones cuando existen otras alternativas. R dispone de una característica que otros lenguajes de programación no disponen, que es conocida como [vectorización](#) y que ya hemos visto con anterioridad.

Además, puesto que R soporta el paradigma de programación funcional existen la posibilidad de hacer uso de funciones en lugar de construcciones imperativas que hemos visto en esta sección y de la vectorización.

Este conjunto de funciones pertenecen la familia de funciones `apply` y las del paquete `purr`.

Vectorización

Como ya hemos visto con anterioridad, la vectorización nos permite realizar operaciones elemento a elemento en vectores y matrices.

Además, deberíamos saber a estas alturas que la estructura de datos elemental en R es el vector. Así pues, una colección de números es un vector numérico.

Si combinamos dos vectores de la misma longitud, obtenemos una matriz. Que como ya hemos visto podemos hacerlo vertical u horizontalmente, mediante el uso de diferentes instrucciones R. Es por eso, que en R una matriz es considerada como una colección de vectores verticales o horizontales. Lo dicho hasta aquí supone que, podemos vectorizar operaciones repetitivas en vectores.

De ahí que, la mayoría de las construcciones iterativas que hemos visto en los ejemplos de esta sección pueden realizarse en R por medio de la vectorización.

Sirva de ejemplo la suma de dos vectores `v1` y `v2` en un vector `v3`, la cual puede realizarse elemento a elemento mediante un bucle `for` como:

```
n <- 4
v1 <- c(1, 2, 3, 4)
v2 <- c(5, 6, 7, 8)
v3 <- vector(mode = "integer", length = length(n))

for (i in 1:n) {
  v3[i] <- v1[i] + v2[i]
}
v3
## [1] 6 8 10 12
```

Si bien, podemos usar como alternativa la vectorización nativa de R:

```
v3 = v1 + v2
v3
## [1] 6 8 10 12
```

El Conjunto de Funciones `apply`

La familia `apply` pertenece al paquete base R y esta formado por un conjunto de funciones que nos permiten manipular una selección de elementos en matrices, arrays, listas y dataframes de forma repetitiva.

Es decir, estas funciones nos permiten iterar por los elementos sin tener la necesidad de utilizar explícitamente una construcción iterativa. Estas funciones toman como entrada una lista, matriz o array y aplican esta función a cada elemento. Esta función podría ser una función de agregación, como por ejemplo la media, u otra función de transformación o selección.

La familia `apply` esta compuesta de las funciones:

- `apply`
- `lapply`
- `sapply`
- `vapply`
- `mapply`
- `rapply`
- `tapply`

El paquete `purrr`

El paquete `purrr` forma parte del ecosistema `tidyverse` y esta compuesto de un conjunto de funciones que aprovechan el paradigma de [programación funcional](#) de R, proporcionando un conjunto completo y consistente de herramientas para trabajar con funciones en listas y vectores.

Instalación

```
# La manera mas facil de conseguir `purrr` es instalar el ecosistema
# tidyverse
install.packages("tidyverse")

# Alternativamente, podemos instalar solo purrr:
install.packages("purrr")
```

El patrón de iterar por un vector, realizando una operación en cada elemento y calcular el resultado es tan común que el paquete `purrr` proporciona una familia de funciones para llevar a cabo esta tarea. Existe una función para cada tipo de salida:

- `map()` devuelve una lista.

- `map_lgl()` devuelve un vector lógico.
- `map_int()` devuelve un vector de enteros.
- `map_dbl()` devuelve un vector de reales.
- `map_chr()` devuelve un vector de tipo carácter.

Cada función toma un vector como entrada, aplica una función a cada elemento, y devuelve un vector de la misma longitud y con los mismos nombres de la entrada. El tipo de vector es determinado por el sufijo en la función `map`.

En este curso y en un [módulo posterior](#), trataremos en profundidad el uso del paquete `purrr` que hace uso de la programación funcional en las iteraciones en oposición al paradigma de la iteración imperativa y que hemos tratado en este capítulo. Una vez dominemos las funciones en el paquete `purrr`, seremos capaces de resolver la gran mayoría de algoritmos que impliquen la iteración con menos código, más legible y con menos errores.

Uso

El siguiente ejemplo sirve para demostrar las diferentes alternativas que disponemos para realizar un [algoritmo iterativo](#) mediante las herramientas que hemos visto en este capítulo.

Consideremos que queremos calcular el cuadrado de cada elemento en una secuencia de enteros del 1 a `n`. La primera solución pasa por utilizar una construcción iterativa:

```
n <- 5
res <- rep(NA_integer_, n)
for (i in seq_len(n)) {
  res[i] <- i^2
}
res
## [1]  1  4  9 16 25
```

La segunda opción es por medio de la vectorización:

```
n <- 5
seq_len(n)^2
## [1]  1  4  9 16 25
```

En tercer lugar, mediante `sapply`:

```
n <- 5
sapply(1:n, function(x) x^2)
## [1]  1  4  9 16 25
```

Por último, mediante `purrr::map()`:

```
n <- 5
map_dbl(1:n, function(x) x^2)
## [1] 1 4 9 16 25
```

En este ejemplo por la sencillez del caso las dos últimas alternativas no son necesarias y la correcta sería hacerlo mediante vectorización. Pero en estructuras de datos y funciones mas complejas optaríamos por cualquiera de las dos últimas opciones.

En este curso y en un [módulo posterior](#), trataremos en profundidad el uso del paquete `purrr` que hace uso de la programación funcional en las iteraciones en oposición al paradigma de la iteración imperativa y que hemos tratado en este capítulo. Una vez dominemos las funciones en el paquete `purrr`, seremos capaces de resolver la gran mayoría de algoritmos que impliquen la iteración con menos código, mas legible y con menos errores.

Consejos para el uso de Bucles en R

- Siempre que sea posible hemos de poner la menor cantidad de instrucciones dentro de una estructura iterativa. Puesto que, todas las instrucciones dentro de un bucle se repiten varios ciclos y quizás no sea necesario.
- Hemos de tener cuidado con el uso de `repeat`, asegurándonos que definimos de forma explícita una condición para finalizar la estructura, de lo contrario nos encontraremos ante una iteración infinita.
- Tratar de usar como alternativa a los bucles anidados la [recursividad en funciones](#). Es decir, es mejor el uso de una o mas llamadas a funciones dentro del bucle a que este sea demasiado grande.
- La peculiar naturaleza de R nos sugiere no usar las construcciones iterativas para todo, cuando disponemos de otras alternativas como la vectorización, la familia `apply` y el paquete `purrr`.

Funciones

Este capítulo nos introduce el concepto de función desde el punto de vista de un programador R e ilustra el rango de acción que las funciones tienen dentro del código R.

Objetivos

Después de leer este capítulo, deberíamos:

- Conocer el concepto de [función](#).
- Saber que son las [funciones en R](#).
- Ser capaces de [crear](#) nuestras funciones.
- Utilizar RStudio para crear, guardar y [ver](#) nuestras funciones.
- [Llamar](#) a funciones definidas en otros scripts.
- Comprender las [llamadas en funciones anidadas](#) en R.
- Entender que son los [argumentos](#) y sus valores por defecto.
- Saber lo que son y como utilizar las [funciones anónimas](#) en R.
- Conocer que R soporta el [paradigma de programación funcional](#).

¿Qué es una Función?

En programación, usamos las funciones conocidas también como subrutinas, procedimientos, métodos etc., para contener un conjunto de instrucciones que queremos usar repetidamente o que, a causa de su complejidad, es mejor que estén independientes en un subprograma y llamarlas cuando sea necesario.

Dicho de otra manera, una función es una pieza de código escrita para llevar a cabo una tarea específica. A su vez, pueden o no aceptar argumentos y es posible que devuelvan o no uno o mas valores.

De hecho, existen varias definiciones formales de lo que es una función que abarcan desde las matemáticas hasta las ciencias de la computación. De forma general, sus argumentos constituyen la entrada y los valores de retorno la salida.

Funciones en R

En R, de acuerdo con la documentación base, definimos una función con la siguiente construcción:

```
function(lista_argumentos) {  
    cuerpo_funcion  
}
```

dónde el código situado dentro de las llaves constituye el *cuerpo* de la función.

Nota que cuando usamos funciones definidas, lo único que debemos conocer es la `lista_argumentos` y gestionar el valor o valores de retorno, si los hubiere.

Funciones Definidas por el Usuario

En el caso que necesitemos llevar a cabo una tarea en particular y deseemos crear nuestra propia función usaremos la siguiente plantilla:

```
function.name <- function(argumentos) {  
    computacion_en_los_argumentos  
    otro_codigo  
}
```

Así que, en la mayoría de los casos, una función tiene un nombre, argumentos usados como entrada a la función, dentro de `()` a continuación de la palabra reservada "function"; un cuerpo, que se trata de una instrucción o conjunto de instrucciones dentro de las llaves `{}`, que llevan a cabo la tarea en particular; y puede tener uno o mas valores de retorno en la salida de la función.

Además, definiremos las funciones de forma similar a las variables con la ayuda del operador de asignación de la forma:

```
nombre_funcion <- function(argumentos) {cuerpo}
```

Recuerda que existen funciones que no disponen de nombres y que son conocidas como [funciones anónimas](#) y que analizaremos en un apartado posterior.

¿Cómo podemos ver nuestras funciones R en RStudio?

LLamar Funciones R en otros Scripts

Llamada a Funciones Anidadas en R

Argumentos y Valores por Defecto

Funciones Anónimas en R

Programación Funcional en R

Resumen

Paquetes

Los paquetes en R son colecciones de funciones y conjunto de datos desarrollados por la comunidad. Estos incrementan la potencialidad de R mejorando las funcionalidades base en R, o añadiendo de nuevas. Por ejemplo, en el campo de Ciencia de los Datos cuando trabajamos con `data.frames`, probablemente usaremos los paquetes `dplyr` o `data.table`, dos de los paquetes mas populares en la comunidad.

Supongamos que, deseamos realizar un procesamiento de lenguaje natural de textos en Coreano, extrayendo datos meteorológicos desde Internet, con toda seguridad existen paquetes en R para efectuar el análisis. Actualmente, el repositorio oficial [CRAN](#) recoge cerca de 10.000 paquetes publicados y, además existen muchos mas publicados en Internet.

Objetivos

Después de leer este capítulo, deberíamos:

- Conocer lo básico del los paquetes en R: [que son los paquetes](#) y porqué deberíamos incorporar su uso en el trabajo con R y [dónde encontrarlos](#).
- Ser capaces de instalar y usar paquetes: cómo podemos [instalar paquetes](#) desde CRAN, servidores espejo CRAN, Bioconductor o GitHub. Además, aprenderemos el uso de la función `install.packages()`, la manera de [actualizarlos](#), [eliminarlos](#), etc. Así mismo, estudiaremos las diferentes [interfaces](#) para su instalación, [carga](#) y [descarga](#) en el entorno de trabajo. Por último, aprenderemos la [diferéncia entre un paquete y una librería](#).
- Por otro lado, veremos como utilizar la documentación, los archivos DESCRIPTION y [otras fuentes de documentación](#).
- Por último, comentaremos cómo escoger el [paquete correcto](#) para nuestro análisis.

¿Qué es un Paquete?

Antes de examinar que son los paquetes, permitidme introducir algunas definiciones. Un paquete es el modo apropiado de organizar nuestro trabajo, si lo que deseamos es compartirlo con otros usuarios. Típicamente, un paquete incluye código (no solamente código R), documentación para el uso del paquete y funciones, y conjuntos de datos.

La información básica sobre un paquete es proporcionada en el archivo **DESCRIPTION**, en el que encontraremos que utilidad tiene el paquete, el autor, la versión, fecha, tipo de licencia y las dependencias del paquete.

No sólo encontraremos el archivo **DESCRIPTION** en repositorios como CRAN sino también podemos acceder a la descripción del paquete desde R con la función `packageDescription("package")`, mediante la documentación del paquete con la ayuda de `help(package = "stats")` o en línea (del inglés, online) en el repositorio del paquete.

Por ejemplo, para acceder a la documentación del paquete `stats` lo haríamos del siguiente modo:

```
packageDescription("stats")
help(package = "stats")
```

¿Qué son los Repositorios?

Un repositorio es el lugar dónde están alojados los paquetes y desde el cuál podemos descargarlos. Aunque nosotros o nuestra organización disponga de un repositorio local, usualmente los paquetes son accesible online para todo el mundo. Los repositorios mas populares de paquetes R son:

- **CRAN**: el cual es el repositorio oficial compuesto de un conjunto de servidores web y ftp mantenidos por la comunidad R a lo largo de todo el mundo. Está coordinado por la fundación R, y para que un paquete pueda ser publicado en el mismo, necesita pasar por diferentes pruebas para asegurar que el paquete cumple con las políticas de CRAN. Podemos encontrar mas detalles al respecto en el siguiente [enlace](#).
- **Bioconductor**: se trata de un repositorio específico para bioinformática. Como CRAN, tiene sus propias [políticas de publicaciones y procesos de revisión](#), disponiendo de una comunidad muy activa que proporciona numerosas conferencias y encuentros a lo largo del año.
- **Github**: a pesar que no es específico para R, github es con toda seguridad el repositorio mas popular para la publicación de proyectos **open source** (del inglés, código abierto). Su popularidad procede del espacio ilimitado que proporciona para el alojamiento de proyectos open source, la integración con git (un software de control de versiones) y, la facilidad de compartir y colaborar con otras personas. Se podría objetar que a pesar de que no proporciona procesos de control, sin embargo, será uno de los repositorios que utilizaremos con más frecuencia siempre y cuándo conozcamos la persona/s que publica el paquete.

¿Cómo Instalar un Paquete R?

Instalación de Paquetes desde Cran

La forma de instalar un paquete R depende de dónde este localizado. Con esto queremos decir que, dependerá del repositorio en dónde este publicado. La forma mas común es usar el repositorio CRAN, sirva de ejemplo la siguiente instrucción para la instalación del paquete `vioplot` :

```
install.packages("vioplot")
```

Después de ejecutar la instrucción anterior recibiremos una serie de mensajes en nuestra pantalla. Esto depende del sistema operativo que usemos, las dependencias y si el paquete es instalado correctamente.

Pasemos a analizar la salida por pantalla de la instalación de `vioplot` , algunos de los mensajes que podemos encontrarnos son los siguientes:

```
Installing package into 'C:/Users/Ruben/Documents/R/win-library/3.4'  
(as 'lib' is unspecified)  
also installing the dependency 'sm'
```

Figura1 - Directorio instalación

El mensaje anterior nos indica el directorio donde será instalado nuestro paquete, podemos indicar un directorio diferente mediante el parámetro `lib` .

```
trying URL 'https://cran.rstudio.com/bin/windows/contrib/3.4/sm_2.2-5.4.zip'  
Content type 'application/zip' length 782714 bytes (764 KB)  
downloaded 764 KB
```

Figura2 - Origen y tamaño del paquete

La información previa nos indica el origen y el tamaño del paquete. Obviamente dependerá del servidor espejo CRAN que hayamos seleccionado.

```
package 'sm' successfully unpacked and MD5 sums checked
package 'vioplot' successfully unpacked and MD5 sums checked
```

Figura3 - Mensajes de instalación

Estos son los mensajes de la instalación, el código fuente, ayuda, tests y finalmente un mensaje indicando que la instalación se ha realizado satisfactoriamente.

```
The downloaded binary packages are in
      c:\Users\Ruben\AppData\Local\Temp\RtmpSkDh40\downloaded_packages
```

Figura4 - Directorio temporal

La última pieza de información nos indica en que directorio se encuentran los archivos originales del paquete. Esta información no es relevante para el uso del paquete, puesto que estos son copiados por defecto en una carpeta temporal.

Hay que mencionar, además que si deseamos instalar mas de un paquete a la vez, tan sólo tenemos que pasar como primer argumento a la función `install.packages()` un vector de caracteres:

```
install.packages("vioplot", "MASS")
```

Instalación de Paquetes desde Servidores Espejo de CRAN

Como hemos mencionado CRAN es una red de servidores (cada uno de ellos conocidos como "mirror", del inglés espejo), así pues tenemos la posibilidad de especificar el servidor que deseamos utilizar. En RStudio, el **mirror** será seleccionado por defecto.

Ahora bien, también tenemos la posibilidad de seleccionar el mirror mediante la función `chooseCRANmirror()`, o directamente dentro de la función `install.packages()` por medio del argumento `repos`. En el siguiente [enlace](#) o con la ayuda de la función `getCRANmirrors()` podemos obtener una lista de los mirros disponibles.

El siguiente ejemplo sirve para usar el mirror de la Red Nacional Española de Investigación (Madrid):

```
install.packages("vioplot", repos = "https://cran.rediris.es/")
```

Instalación de Paquetes Bioconductor

En el caso de Bioconductor, el modo estandar de instalar un paquete es ejecutar el siguiente script:

```
source("https://bioconductor.org/biocLite.R")
```

La instrucción anterior instalará en nuestro equipo las funciones necesarias para la instalación de paquetes bioconductor, como por ejemplo la función `biocLite()`. Si deseamos instalar los paquetes básicos de Bioconductor, podemos hacerlo de la manera siguiente:

```
biocLite()
```

No obstante, si solamente estamos interesados en uno o varios paquetes en particular, podemos hacerlo como se muestra a continuación:

```
biocLite(c("GenomicFeatures", "AnnotationDbi"))
```

Instalación de Paquetes con `devtools`

Como hemos visto hasta ahora, cada repositorio tiene su forma particular de instalar un paquete, en el caso de que regularmente utilicemos diferentes fuentes en la instalación de nuestros paquetes este método puede llegar a convertirse en una labor un poco pesada. Un modo mas eficiente es mediante el uso del paquete `devtools` que nos simplificará esta tarea, puesto que contiene funciones específicas para cada repositorio, incluyendo CRAN.

En primer lugar, tendremos que instalar el paquete `devtools` como hemos visto en un apartado anterior:

```
install.packages("devtools")
```

Una vez que tenemos instalado `devtools`, haremos uso de las funciones para la instalación de otros paquetes. Las opciones son las siguientes:

- `install_bioc()` desde [Bioconductor](#),
- `install_bitbucket()` desde [Bitbucket](#),
- `install_cran()` desde [CRAN](#),
- `install_git()` desde un repositorio [git](#),
- `install_github()` desde [GitHub](#),
- `install_local()` desde un archivo alojado en nuestro equipo,

- `install_svn()` desde un repositorio SVN,
- `install_url()` desde una URL, y
- `install_version()` para una versión específica de un paquete de CRAN.

Por ejemplo, para instalar el paquete `babynames` desde su repositorio github podemos hacerlo como mostramos a continuación:

```
devtools::install_github("hadley/babynames")
```

Actualizar, Eliminar y Comprobar Paquetes Instalados

En esta sección, encontraremos unas pocas funciones que nos permitirán la gestión de la colección de paquetes instalados en nuestro equipo.

- Para comprobar que paquetes tenemos instalados en nuestro equipo, usaremos:

```
installed.packages()
```

- Si nuestro objetivo es eliminar un paquete, podemos hacerlo como en el siguiente ejemplo:

```
remove.packages("vioplot")
```

- Para comprobar que paquetes necesitan ser actualizados usaremos:

```
old.packages()
```

- A su vez para actualizar todos los paquetes lo haremos mediante una llamada a la función:

```
update.packages()
```

- Sin embargo, para actualizar un solo paquete, usaremos de nuevo la instrucción:

```
install.packages("vioplot")
```

Interfaces Gráficas para la Instalación de Paquetes

En RStudio en Tools -> Install Package se nos abrirá un cuadro de dialogo para introducir el paquete que deseamos instalar:

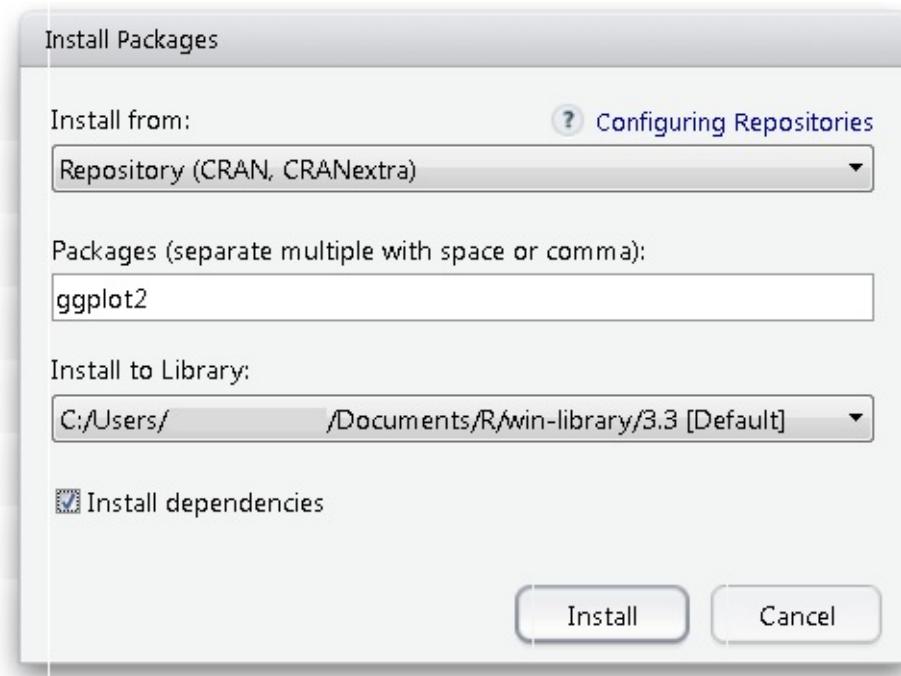


Figura5 - Instalación Paquetes RStudio

Carga de Paquetes

Una vez tenemos instalado un paquete, estamos en disposición de hacer uso de sus funciones. Si tan solo necesitamos hacer un uso esporádico de unas pocas funciones o datos del paquete podemos acceder con la notación `nombrepaqute::nombrefuncion()`. Por ejemplo, siguiendo con el ejemplo anterior del paquete `babynames`, para explorar uno de sus datasets podemos hacer uso de la siguiente instrucción:

```
babynames::births
## # A tibble: 119 x 2
##   year   births
##   <int>   <int>
## 1 1909 2718000
## 2 1910 2777000
## 3 1911 2809000
## 4 1912 2840000
## 5 1913 2869000
## 6 1914 2966000
## 7 1915 2965000
## 8 1916 2964000
## 9 1917 2944000
## 10 1918 2948000
## # ... with 109 more rows
```

RECUERDA que para acceder a las funciones y conjuntos de datos que contiene un paquete, podemos hacerlo mediante:

```
help(package = "babynames")
```

Por otro lado, si hacemos un uso intensivo de un paquete, la forma mas eficiente de trabajar es cargándolo en memoria. Esto lo conseguiremos mediante el uso de la función `library()`.

```
library(babynames)
```

Después de cargar el paquete en memoria, ya no será necesario hacer uso de la notación `nombrepaquete::funcion()`, y podremos hacer uso de sus funciones y datos como cualquier paquete base de R:

```
births
## # A tibble: 119 x 2
##   year   births
##   <int>   <int>
## 1 1909 2718000
## 2 1910 2777000
## 3 1911 2809000
## 4 1912 2840000
## 5 1913 2869000
## 6 1914 2966000
## 7 1915 2965000
## 8 1916 2964000
## 9 1917 2944000
## 10 1918 2948000
## # ... with 109 more rows
```

NOTA: que el argumento de la función `install.packages()` es un vector de tipo carácter y requiere de los nombres de los paquetes entrecomillados, mientras que la función `library()` acepta tanto un vector de tipo carácter como el nombre sin comillas.

De manera semejante, podemos hacer uso de la función `require()` para cargar un paquete en memoria, sin embargo la diferencia es que no lanzará un error si el paquete no está instalado. Es por eso, que recomendamos hacer uso de esta función con precaución.

Cómo Cargar Varios Paquetes al mismo Tiempo

A pesar de que podemos pasar como argumento un vector de caracteres a la función `install.packages()` para instalar varios paquetes de una sola vez, en el caso de la función `library()`, esto no es posible.

Cómo Descargar un Paquete

Para descargar de la memoria un cierto paquete podemos hacer uso de la función `detach`. Su uso es el siguiente:

```
detach("babynames", unload = TRUE)
```

Diferencia entre un Paquete y una Librería

En el caso de la función `library()`, en muchas ocasiones existe la confusión entre un paquete y una librería, y podemos encontrarnos con gente llamando "librerías" a los paquetes.

Una cosa es la función `library()` usada para cargar un paquete, y que se refiere al lugar en dónde el paquete es localizado, habitualmente una carpeta en nuestro ordenador, y otra un paquete que es una colección de funciones y datos empaquetados de forma conveniente.

Dicho de otra manera, y en palabras de [Hadley Wickham](#), científico de datos en RStudio, y creador de la gran mayoría de paquetes del ecosistema [tidyverse](#):



Ian Lyttle @ijlyttle

Dec 8, 2014

Why are they called libraries when they are actually packages? Or is it the other way around? [#rstatspic.twitter.com/YpA1lqyYpq](#)



Hadley Wickham

@hadleywickham

Follow

@ijlyttle a package is like a book, a library is like a library; you use library() to check a package out of the library #rsats

3:34 PM - Dec 8, 2014

Q 2 T 11 L 22



Figura6 - Diferencia entre Paquete y Libreria

De igual manera, para recordar la diferencia es ejecutar la función `library()` sin argumentos. Esto nos proporcionará una lista de paquetes instalados en diferentes librerías en nuestro ordenador:

```
library()
```

Diferentes Fuentes de Documentación y Ayuda

Como ya sabemos de secciones anteriores, el archivo DESCRIPTION contiene información básica del paquete, y a pesar de que esta información es de gran utilidad, es probable que no sea conveniente para nuestro análisis. De ahí que necesitemos consultar dos fuentes mas de información: los archivos de ayuda y las vignettes.

Los Archivos de Ayuda

En primer lugar, como primer recurso para obtener información de un paquete es mediante los comandos `?()` y `help()`. Recordemos que podemos conseguir información general del paquete mediante `help(package = "packagename")`, pero además cada función puede ser consultada individualmente mediante `help("nombre de la función")` o `help(function, package = "package")` si el paquete no ha sido cargado. Estas instrucciones nos mostrarán la descripción de la función y sus argumentos acompañados de ejemplos de utilización.

Por ejemplo, para obtener el archivo de ayuda de la función `vioplot` del paquete `vioplot` podemos hacerlo mediante la siguiente instrucción:

```
help(vioplot, package = "vioplot")
```

Nota: Podemos utilizar otra alternativa para inspeccionar un paquete cargado y es con la ayuda de la función `ls()` como se muestra a continuación:

```
library(babynames)
ls("package:babynames")
## [1] "applicants" "babynames"   "births"       "lifetables"
```

Vignettes

Otra fuente de ayuda incluida en la mayoría de paquetes son las vignettes. Estos documentos son utilizados por los autores para mostrarnos de una forma mas detallada las funcionalidades del paquete. El uso de las vignettes es un método útil para conocer la utilización del paquete, es por eso que se trata de un punto de partida antes de comenzar con nuestro propio análisis.

La información de las vignettes de un paquete están disponibles en el archivo DOCUMENTATION localmente o en línea. Además podemos obtener una lista de vignettes incluidas en nuestros paquetes instalados con la función `browseVignettes()`, en caso de que únicamente deseemos consultar las vignettes de un paquete pasaremos como argumento a la función el nombre del mismo: `browseVignettes(package = "packagename")`. En ambos casos, una ventana del navegador se abrirá para que podamos fácilmente explorar por el documento.

En el caso que, optemos por permanecer en la línea de comandos, la instrucción `vignette()` nos mostrará una lista de vignettes, `vignette(package = "packagename")` las vignettes incluidas en el paquete, y por último una vez hemos localizado la vignette de nuestro interés podemos explorarla mediante el uso de `vignette("vignettename")`.

Por ejemplo, uno de los paquetes mas utilizados para la visualización de datos es `ggplot2` y que trataremos con profundidad en apartados posteriores. Es probable que ya lo tengamos instalado en nuestro computador, en caso contrario es una buena oportunidad para poner en práctica lo que hemos aprendido hasta el momento.

Asumiendo que ya disponemos del paquete `ggplot2` instalado, podemos consultar las vignettes que incluye mediante la siguiente instrucción:

```
vignette(package = "ggplot2")
```

Si ejecutamos la instrucción anterior podemos comprobar que existen dos vignettes disponibles para `ggplot2`, "ggplot2-specs" y "extending-ggplot2". Podemos explorar la primera mediante la siguiente instrucción:

```
vignette("ggplot2-specs")
```

En RStudio, el documento será mostrado en la pestaña de ayuda, mientras que en la interfaz RGui o la línea de comandos se nos mostrara en el navegador web.

En este [enlace](#), podemos encontrar mas opciones para obtener ayuda en R.

Escoger el Paquete Correcto para Nuestro Análisis

En conclusión, hasta el momento hemos aprendido las herramientas para instalar y obtener la mayoría de paquetes para R, sin embargo todavía queda una cuestión en el aire: dónde podemos encontrar los paquetes que necesitamos.

El modo habitual de conocer los paquetes es aprendiendo R, en la mayoría de tutoriales, libros y cursos los paquetes mas populares son mencionados. Este curso puede ser un buen ejemplo: el módulo de [importar datos](#) te enseñara todo lo necesario en el uso de los paquetes `readr`, `readxl` y `haven`, el módulo de [visualización de datos](#) trata sobre `ggplot2`, y un largo etcétera.

Sin embargo, si nos encontramos ante una situación específica y no sabemos por donde empezar, por ejemplo como introduce en la introducción del capítulo en la que estábamos interesados en analizar texto en Coreano.

Lo anterior no quiere decir que no dispongamos de otros medios para buscar nuestros paquetes. Permitidme explorar algunas alternativas.

Una posibilidad es navegar por las categorías de los paquetes en CRAN, con la ayuda de [CRAN task views](#). En esta vista podremos navegar por los paquetes agrupados por temas o categorías basado en su funcionalidad.

CRAN Task Views

Bayesian	Bayesian Inference
ChemPhys	Chemometrics and Computational Physics
ClinicalTrials	Clinical Trial Design, Monitoring, and Analysis
Cluster	Cluster Analysis & Finite Mixture Models
DifferentialEquations	Differential Equations
Distributions	Probability Distributions
Econometrics	Econometrics
Environmetrics	Analysis of Ecological and Environmental Data
ExperimentalDesign	Design of Experiments (DoE) & Analysis of Experimental Data
ExtremeValue	Extreme Value Analysis
Finance	Empirical Finance
FunctionalData	Functional Data Analysis
Genetics	Statistical Genetics
Graphics	Graphic Displays & Dynamic Graphics & Graphic Devices & Visualization
HighPerformanceComputing	High-Performance and Parallel Computing with R
MachineLearning	Machine Learning & Statistical Learning
MedicalImaging	Medical Image Analysis
MetaAnalysis	Meta-Analysis

Figura 7 - CRAN Task Views

Volviendo al ejemplo del texto en Coreano, podemos con facilidad encontrar el paquete que necesitamos navegando al enlace Natural Language Processing. Desde este enlace, podemos buscar en la página hasta encontrar el paquete para realizar el procesamiento de lenguaje coreano, o bien mediante el atajo de teclado CTRL + F introduciendo un criterio de búsqueda en la barra que nos aparecerá.

Otra alternativa es utilizar la [RDocumentation](#), un agregador para documentación de paquetes R de fuentes como CRAN, BioConductor y GitHub, el cual nos ofrece un barra de búsqueda directamente en la página principal.

Siguiendo con el ejemplo del texto en Coreano, a medida que vamos introduciendo texto en el cuadro de texto se nos muestran algunos resultados:

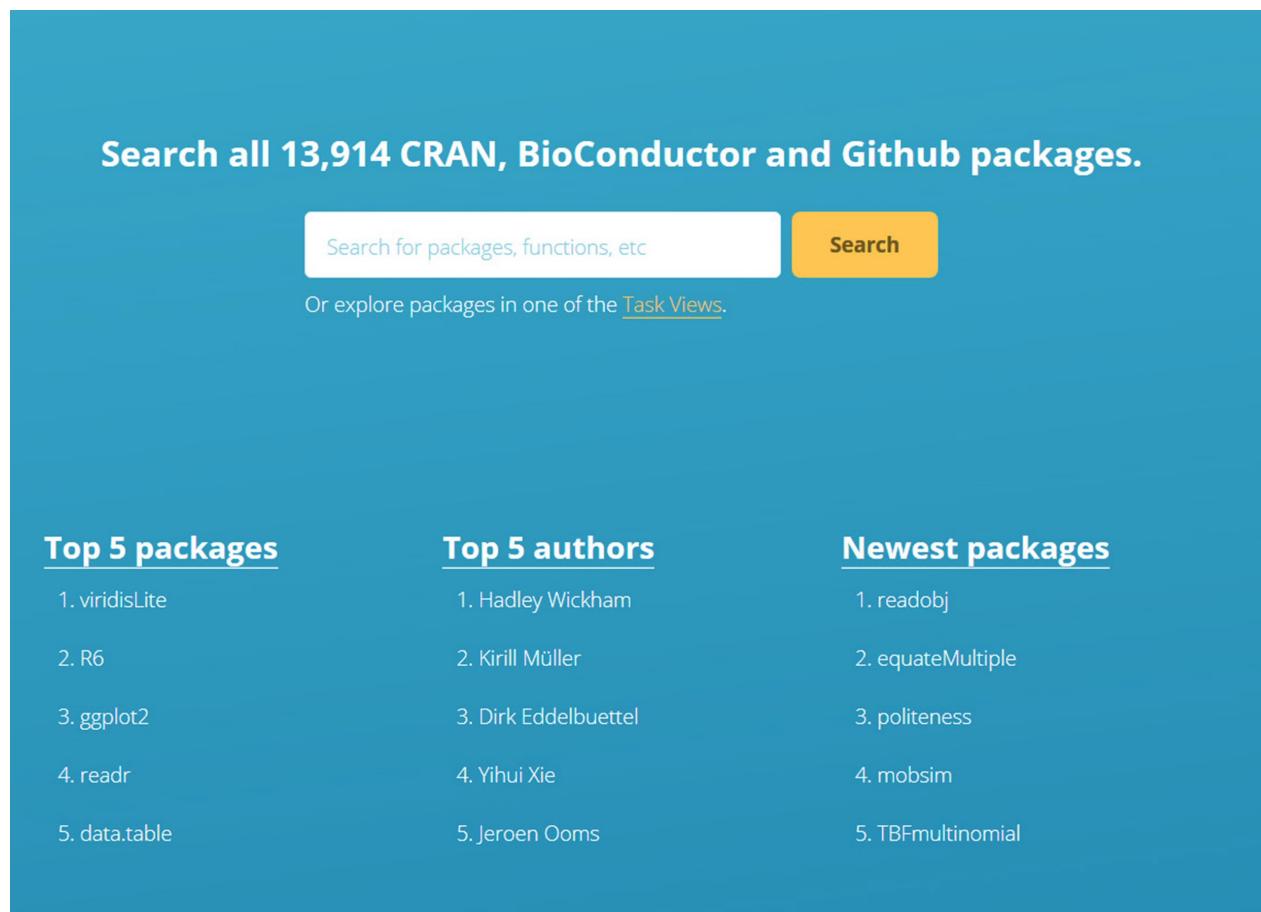


Figura8 - RDocumentation

Pero introduzcamos la palabra "korean" para realizar una búsqueda completa y presionemos en el botón "Search". Podemos observar que se nos mostrara dos columnas con los resultados: los paquetes en la izquierda y las funciones en la derecha.

The screenshot shows the R Documentation search interface. The search term 'korean' is entered in the search bar. The results are categorized into 'Packages' and 'Functions'. Under 'Packages', there are four entries: 'KoNLP v0.80.1' by Heewon Jeon, 'Rtextrankr v1.0.0' by Junghoon Seo, 'translation.ko v0.0.1.5.2' by Chel Hee Lee, and 'moonBook v0.1.3' by KeonWoong Moon. Each package entry includes a brief description and its popularity rank (e.g., 57th percentile). Under 'Functions', there are two entries: 'l10n_info' and 'strtrim'. Each function entry includes its documentation link and a brief description.

Category	Name	Description	Rank
Packages	KoNLP v0.80.1	POS Tagger and Morphological Analyzer for Korean text based research. It provides tools for corpus linguistics research such as Keystroke converter, Hangul automata, Concordance, and Mutual Information. It also provides a convenient interface for NLP tasks like sentiment analysis and named entity recognition.	12,963 results 57th Percentile
	Rtextrankr v1.0.0	Reorder sentences for Korean text using TextRank algorithm.	
	translation.ko v0.0.1.5.2	R version 2.1.0 and later support Korean translations of program messages. The continuous efforts have been made by The R Documentation files are licensed under the General Public License, version 2 or 3. This means that the code is free to use and can be modified and distributed.	
	moonBook v0.1.3	Several analysis-related functions for the book entitled "R statistics and graph for medical articles" (written in Korean), version 1, by Keon-Woong Moon with Korean demographic data with several plot functions.	
Functions	l10n_info	Report on localization information.	
	strtrim	Trim Character Strings to Specified Display Widths	

Figura9 - Resultado de la búsqueda

Centrándonos en la columna de paquetes, para cada resultado obtendremos el nombre de el paquete, junto a un enlace para una información mas detallada, el nombre del autor, al que también podemos acceder mediante el enlace para obtener otros paquetes del mismo autor, una descripción de el paquete con el criterio de búsqueda resaltado e información sobre la popularidad del paquete.

Search Results for korean

Search through all versions

Packages

12,963 results

[KoNLP v0.80.1](#) by [Heewon Jeon](#)

57th

POS Tagger and Morphological Analyzer for **Korean** text based research. It provides tools for corpus linguistics research such as Keystroke converter, Hangul automata, Concordance, Percentile and Mutual Information. It also provides a convenient interface.

[Rtextrankr v1.0.0](#) by [Junghoon Seo](#)

Reorder sentences for **Korean** text using TextRank algorithm.

[translation.ko v0.0.1.5.2](#) by [Chel Hee Lee](#)

R version 2.1.0 and later support **Korean** translations of program messages. The continuous efforts have been made by The R Documentation files are licensed under the General Public License, version 2 or 3. This means that the p

[moonBook v0.1.3](#) by [KeonWoong Moon](#)

Several analysis-related functions for the book entitled "R statistics and graph for medical articles" (written in **Korean**), version 1, by Keon-Woong Moon with **Korean** demographic data with several plot functions.

Figura10 - Columna paquetes

Respecto a la popularidad del paquete, es un indicador relevante puesto que la búsqueda ordena los paquetes por descarga, de un modo que mejora la relevancia de los resultados.

Dado que, parece ser que el paquete **KoNLP** cubre nuestras necesidades, después de hacer clic en su nombre, se nos mostrara la siguiente información:

- Una cabecera con el nombre del paquete, el autor, la versión, una opción para obtener versiones anteriores, el número de descargas y un enlace a la página de RDocumentation.

- Una descripción del paquete.
- La lista de las funciones incluidas en el paquete, a las que podemos acceder haciendo clic en las mismas para obtener información detallada del uso de la función. Además, se nos mostrará una caja de búsqueda para un rápido acceso a la función deseada.
- Un gráfico con la evolución de el número de descargas.
- Los detalles del paquete con la información del archivo DESCRIPTION.

Hay que mencionar, además que RDocumentation no es únicamente un motor de búsqueda, puesto que nos ofrece una serie de características para buscar y aprender sobre paquetes y funciones R:

- De igual manera que las task views de CRAN, RDocumentation nos ofrece sus propias [task views](#) que como hemos visto en apartados anteriores, es otro método para explorar paquetes R por temas.
- [Leaderboard](#). Esta página nos muestra información sobre los paquetes y desarrolladores más populares, los paquetes más recientes y actualizaciones, y tres gráficos con la distribución de paquetes descargados, las palabras claves más populares y un gráfico de dependencias, en el que podemos comprobar como los paquetes más populares están relacionados.

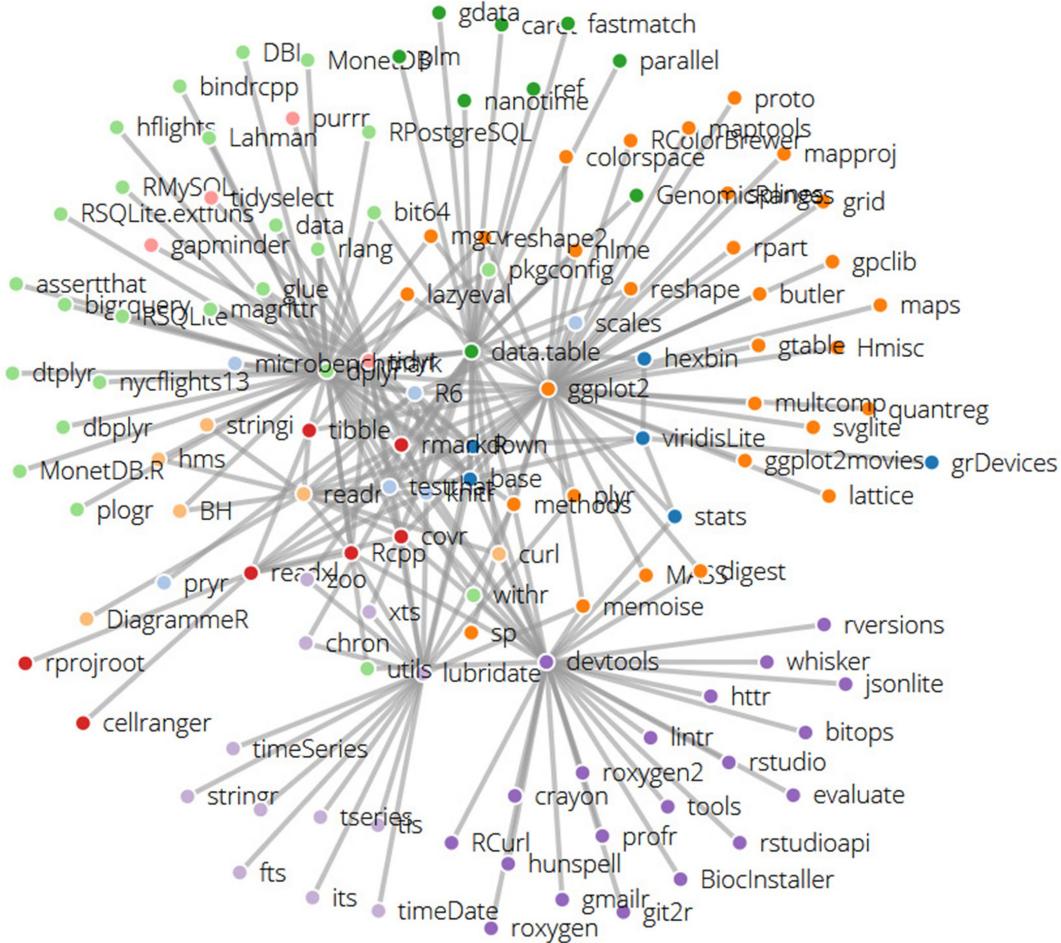


Figura 11 - Gráfico de dependencias

- El paquete [RDocumentation](#). RDocumentation no sólo es una página web sino también un paquete R. Este paquete nos permite incorporar la potencia de RDocumentation en nuestro flujo de trabajo. Una vez este paquete es cargado, la función `help()` abrirá una ventana en RStudio con acceso a la documentación de RDocumentation.

Disponer de RDocumentation en RStudio nos proporciona ventajas comparado al uso de la versión web, puesto que:

- Podemos comprobar la versión del paquete instalado. El panel de ayuda de un paquete proporciona la misma información que la versión web (descargas, descripción, lista de funciones y detalles), además de información sobre la versión instalada del paquete.

A modo de ejemplo, comprobemos el paquete `vioplot` que hemos instalado en un apartado anterior:

```
install.packages("RDocumentation")
library(RDocumentation)
help(package = "vioplot")
```

vioplot v0.2

[Other versions](#) ▾

✓ You have the latest version

2,984 Monthly downloads ➤ 71th Percentile

by [Daniel Adler](#)

<https://www.rdocumentation.org/packages/vioplot>

[Copy](#)

Figura12 - RDocumentation en RStudio

- Además de proporcionarnos la habilidad de instalar y actualizar paquetes directamente desde el panel de ayuda. Por ejemplo, eliminemos el paquete `vioplot` de nuestro equipo y probemos a instalar el paquete desde el panel de ayuda mediante el botón que nos proporciona RDocumentation:

```
remove.packages("vioplot")
help(package = "vioplot")
```

vioplot v0.2

[Other versions](#) ▾

2,984 Monthly downloads ➤ 71th Percentile

by [Daniel Adler](#)

<https://www.rdocumentation.org/packages/vioplot>

[Install](#)

[Copy](#)

Figura13 - Instalación paquetes con RDocumentation

- Ejecutar y proponer ejemplos. El panel de ayuda de las funciones de un paquete proporciona la opción de ejecutar ejemplos con un solo clic en un botón. Además tenemos la posibilidad de incorporar ejemplos en la página de ayuda para que sean utilizados por otros usuarios de R.

```
install.packages("vioplot")
help(vioplot)
```

Examples

```
# box- vs violin-plot
par(mfrow=c(2,1))
mu<-2
si<-0.6
bimodal<-c(rnorm(1000,-mu,si),rnorm(1000,mu,si))
uniform<-runif(2000,-4,4)
normal<-rnorm(2000,0,3)
vioplot(bimodal,uniform,normal)
boxplot(bimodal,uniform,normal)

# add to an existing plot
x <- rnorm(100)
y <- rnorm(100)
plot(x, y, xlim=c(-5,5), ylim=c(-5,5))
vioplot(x, col="tomato", horizontal=TRUE, at=-4, add=TRUE, lty=2, rectCol="gray")
vioplot(y, col="cyan", horizontal=FALSE, at=-4, add=TRUE, lty=2)
```

[Run examples](#)

Documentation reproduced from package vioplot, version 0.2. License: undefined

Figura 14 - Ejecutar ejemplos

Importar Datos en R

Para importar datos en R, obviamente necesitamos disponer de datos. Estos datos pueden estar guardados en un archivo en nuestro computador en Excel, SPSS, u otro tipo de archivo. En este escenario, en el que nuestros datos están guardados localmente, podemos siempre que queramos editarlos cuando deseemos, añadir más datos o modificarlos, preservando las fórmulas que hemos usado para calcular los resultados, etc.

Sin embargo, los datos también podemos encontrarlos en Internet o podemos obtenerlos de otras fuentes.

En es capítulo veremos las diferentes formas de importar datos en R desde diferentes archivos y fuentes.

Comprobación de Nuestros Datos

Antes de conocer como importar nuestros datos en R, es importante comprobar la siguiente lista de verificación:

- Si trabajamos con hojas de cálculo, la primera fila normalmente está reservada para la cabecera, mientras que la primera columna es usada para identificar la observación.
- Debemos de evitar nombres, valores o campos con espacios en blanco, de lo contrario cada palabra será interpretada como una variable, resultando en errores relacionados con el número de elementos por fila en nuestro conjunto de datos.
- Si deseamos concatenar palabras, aconsejamos insertar un guión (-) entre las palabras en lugar de un espacio en blanco.
- Escoger nombres cortos en lugar de largos.
- Evitar usar los siguientes símbolos en los nombres: ?, \$, %, ^, &, *, (,), -, #, ?, , <, >, /, |, \, [,], { y } .
- Borrar cualquier comentario que hayamos insertado en nuestro archivo Excel para evitar columnas extra, de lo contrario valores NA serán introducidos en nuestro archivo.
- Comprobar que cualquier valor desconocido en nuestros datos es indicado como NA .

Preparación del Espacio de Trabajo

Antes de importar datos en R es necesario además preparar nuestro entorno de trabajo en RStudio. Podríamos encontrarnos con un entorno que se encuentra atiborrado con datos y valores, que podemos borrar con la siguiente línea de código:

```
rm(list = ls())
```

La función `rm()` nos permite borrar todos los objetos de un entorno específico. En el ejemplo, pasamos como argumento la función `ls()`. Esta función devuelve un vector de tipo **character** que nos proporciona los nombres de los objetos en el entorno especificado.

A continuación, es importante conocer en qué directorio de trabajo nos encontramos, que podemos comprobar mediante:

```
getwd()
```

Por último, si en el paso anterior confirmamos que nuestros datos no se encuentran en el directorio devuelto por la función `getwd()`, debemos cambiar la ruta al directorio donde se encuentran nuestro conjunto de datos por medio de:

```
setwd("<localización de nuestro dataset>")
```

Lectura de Archivos en Formato Tabular

En este apartado, veremos como importar archivos de texto y Excel a R con el paquete `readr`, el cual es parte del ecosistema `tidyverse`:

```
library(tidyverse)
```

Lectura de Archivos de Texto con `readr`

La mayoría de las funciones del paquete `readr` están destinadas a convertir [archivos en formato tabular](#) en `dataframes`:

* `read_csv()` lee archivos en que las columnas están separadas por comas, `read_csv2()` lee archivos separados por punto y coma (comunes en países donde `,` es usada para separar los decimales en los números), `read_tsv()` lee archivos separados por tabuladores, y `read_delim()` lee archivos separados por cualquier otro tipo de delimitador.

* `read_fwf()` lee archivos con un ancho fijo. Podemos especificar los campos o por su ancho con `fwf_widths()` o su posición mediante `fwf_positions()`. `read_table()` lee un tipo de variación de archivos con un ancho fijo donde las columnas están separadas por espacios en blanco.

Estas funciones comparten la misma sintaxis: una vez hemos dominada una, podemos hacer uso de las otras funciones sin dificultad. Para el resto de este capítulo nos centraremos en `read_csv`. No sólo porque los archivos `csv` son una de las formas mas comunes de almacenamiento de datos, sino también porque una vez entendamos `read_csv()`, podremos con facilidad aplicar nuestros conocimientos al resto de las otras funciones del paquete `readr`.

El primer argumento en `read_csv()` es el mas importante, puesto que se trata de la localización del archivo que deseamos importar:

```
df <- read_csv("data/Water_Right_Applications.csv")
## Parsed with column specification:
## cols(
##   .default = col_character(),
##   WR_DOC_ID = col_integer(),
##   YEAR_APPLIED = col_integer(),
##   CFS = col_double(),
##   GPM = col_double(),
##   DOMESTIC_UNITS = col_integer(),
##   ACRE_FEET = col_double(),
##   ACRE_IRR = col_double(),
##   WRIA_NUMBER = col_integer(),
##   Latitude1 = col_double(),
##   Longitude1 = col_double()
## )
## See spec(...) for full column specifications.
```

Observemos que cuando ejecutamos la función `read_csv` muestra por la consola información relativa al nombre y tipo de cada columna. Esto es una característica muy útil en `readr`.

Podemos pasar un archivo csv en una sola línea de código. Esto es útil para experimentar con `readr` y para crear ejemplos reproducibles que podemos compartir con otros:

```
read_csv("a,b,c
1,2,3
4,5,6")
## # A tibble: 2 × 3
##       a     b     c
##   <int> <int> <int>
## 1     1     2     3
## 2     4     5     6
```

En ambos casos `read_csv()` usa la primera línea como nombre de las columnas, lo que es una convención generalizada. Existen dos situaciones en las que estaríamos interesados en modificar este comportamiento:

1. En ocasiones existen unas pocas líneas de metadatos en la cabecera del archivo. Podemos hacer uso de `skip=n` para descartar las `n` primeras líneas; o usar `comment = "#"` para eliminar todas las líneas que empiezan con `#`.

```
read_csv("La primera linea de metadatos
        La segunda linea de metadatos
        x,y,z
        1,2,3",
        skip = 2)
## # A tibble: 1 x 3
##       x     y     z
##   <int> <int> <int>
## 1     1     2     3
```

```
read_csv("# Un comentario que deseamos eliminar
        x,y,z
        1,2,3", comment = "#")
## # A tibble: 1 x 3
##       x     y     z
##   <int> <int> <int>
## 1     1     2     3
```

1. Los datos podrían no contener columnas con nombres. En esta situación podemos hacer uso del argumento `col_names=FALSE` para indicar a `read_csv()` que no trate la primera linea como cabecera, y en lugar etiquete secuencialmente de `x1` a `xn`:

```
read_csv("1,2,3\n4,5,6", col_names = FALSE)
## # A tibble: 2 x 3
##       x1     x2     x3
##   <int> <int> <int>
## 1     1     2     3
## 2     4     5     6
```

Nota que hemos usado `"\n"` para añadir una nueva linea.

Alternativamente, podemos pasar a `col_names` un vector de caracteres que será usado para los nombres de las columnas:

```
read_csv("1,2,3\n4,5,6", col_names = c("x", "y", "z"))
## # A tibble: 2 x 3
##       x     y     z
##   <int> <int> <int>
## 1     1     2     3
## 2     4     5     6
```

Otro argumento de gran utilidad es `na`, que especifica el valor (o valores) que son usados para representar los valores desconocidos en nuestro archivo:

```
read_csv("a,b,c\n1,2,#N/A", na = "#N/A")
## # A tibble: 1 × 3
##      a     b     c
##   <int> <int> <chr>
## 1     1     2 <NA>
```

Esto es todo lo que necesitamos saber para importar ~75% de archivos csv que nos encontraremos en la práctica. Además, podemos fácilmente adaptar lo que hemos aprendido en esta sección para leer archivos separados por tabuladores con `read_tsv()` y archivos con un ancho fijo con `read_fwf()`.

Por otro lado, cabe mencionar que `readr` proporciona un conjunto de funciones para realizar la tarea inversa, es decir disponemos de un grupo de funciones que nos permiten escribir nuestros datos a archivos.

Escribiremos un objeto `x` a una ruta específica por medio del argumento `path` como:

Archivos CSV

```
write_csv(x, path, na = "NA", append = FALSE, col_names = !append)
```

Archivos con Delimitadores Arbitrarios

```
write_delim(x, path, delim = "", na = "NA", append = FALSE, col_names = !append)
```

CSV para Excel

```
write_excel_csv(x, path, na = "NA", append = FALSE, col_names = !append)
```

Archivos Delimitados por Tabuladores

```
write_tsv(x, path, na = "NA", append = FALSE, col_names = !append)
```

Lectura Archivos Excel con `readxl`

El paquete `readxl` facilita la importación de archivos Excel en R. Comparado con la mayoría de paquetes existentes (e.g. `gdata`, `xlsx`, `xlsReadWrite`) `readxl` no requiere de dependencias externas, por lo tanto es más fácil de instalar y usar en los sistemas

operativos. Como en los apartados anteriores es usado para el trabajo de datos en formato tabular. Este paquete forma parte de la colección de paquetes del ecosistema [tidyverse](#) que comparten la misma filosofía y están diseñados para trabajar de forma conjunta.

Instalación

El modo más fácil de instalar la última versión desde CRAN es instalar el conjunto de paquetes [tidyverse](#).

```
install.packages("tidyverse")
```

Alternativamente, podemos instalar solo `readxl` desde CRAN mediante la siguiente línea de código:

```
install.packages("readxl")
```

O instalar la versión en desarrollo desde GitHub:

```
# install.packages('devtools')
devtools::install_github("tidyverse/readxl")
```

NOTA: Para esta última forma debemos tener instalado el paquete `devtools`.

Uso del paquete `readxl`

En primer lugar tendremos que cargarlo:

```
# Cargamos el paquete
library(readxl)
```

La función `read_excel()` nos permite leer los archivos tanto si son `xls` o `xlsx`:

```
hoja_calculo_xlsx <- read_excel("data/datasets.xlsx", n_max = 6)
hoja_calculo_xlsx
## # A tibble: 6 x 5
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
##       <dbl>      <dbl>      <dbl>      <dbl>   <chr>
## 1       5.1       3.5       1.4       0.2   setosa
## 2       4.9       3.0       1.4       0.2   setosa
## 3       4.7       3.2       1.3       0.2   setosa
## 4       4.6       3.1       1.5       0.2   setosa
## 5       5.0       3.6       1.4       0.2   setosa
## 6       5.4       3.9       1.7       0.4   setosa
```

Observemos que en el siguiente ejemplo a pesar que la extensión del archivo es `.xls` utilizamos la misma función que en el caso anterior:

```
hoja_calculo.xls <- read_excel("data/datasets.xls", n_max = 8)
# Mostramos la hoja de calculo
hoja_calculo.xls
## # A tibble: 8 x 5
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
##       <dbl>      <dbl>      <dbl>      <dbl>   <chr>
## 1       5.1       3.5       1.4       0.2   setosa
## 2       4.9       3.0       1.4       0.2   setosa
## 3       4.7       3.2       1.3       0.2   setosa
## 4       4.6       3.1       1.5       0.2   setosa
## 5       5.0       3.6       1.4       0.2   setosa
## 6       5.4       3.9       1.7       0.4   setosa
## 7       4.6       3.4       1.4       0.3   setosa
## 8       5.0       3.4       1.5       0.2   setosa
```

Podemos obtener los nombres de las hojas en el libro con la ayuda de la función `excel_sheets()`:

```
excel_sheets("data/datasets.xls")
## [1] "iris"     "mtcars"    "chickwts"  "quakes"
```

Es posible seleccionar una hoja de cálculo mediante su nombre o por la posición que ocupa en el libro:

```
read_excel(path = "data/datasets.xls", sheet = "iris", n_max = 8)
```

```
read_excel(path = "data/datasets.xlsx", sheet = 1, n_max = 8)
## # A tibble: 8 x 5
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
##       <dbl>      <dbl>      <dbl>      <dbl>    <chr>
## 1       5.1       3.5       1.4       0.2  setosa
## 2       4.9       3.0       1.4       0.2  setosa
## 3       4.7       3.2       1.3       0.2  setosa
## 4       4.6       3.1       1.5       0.2  setosa
## 5       5.0       3.6       1.4       0.2  setosa
## 6       5.4       3.9       1.7       0.4  setosa
## 7       4.6       3.4       1.4       0.3  setosa
## 8       5.0       3.4       1.5       0.2  setosa
```

El argumento `range` de la función `read_excel` nos permite seleccionar un rango de filas y columnas con la misma nomenclatura que utilizamos en Excel:

```
read_excel(path = "data/datasets.xls", range = "C1:E4")
## # A tibble: 3 x 3
##   Petal.Length Petal.Width Species
##       <dbl>      <dbl>    <chr>
## 1       1.4       0.2  setosa
## 2       1.4       0.2  setosa
## 3       1.3       0.2  setosa
```

Además, podemos especificar la hoja del libro de la cual queremos extraer un rango y con el símbolo lógico de negación `!` excluir celdas y columnas en nuestra selección:

```
read_excel(path = "data/datasets.xls", range = "iris!B1:D5")
## # A tibble: 4 x 3
##   Sepal.Width Petal.Length Petal.Width
##       <dbl>      <dbl>      <dbl>
## 1       3.5       1.4       0.2
## 2       3.0       1.4       0.2
## 3       3.2       1.3       0.2
## 4       3.1       1.5       0.2
```

Por defecto, la función `read_excel` trata los campos vacíos como valores desconocidos `NA`. En caso contrario, debemos especificarlo en el argumento `na`:

```
read_excel(path = "data/datasets.xls", na = "setosa", n_max = 8)
## # A tibble: 8 x 5
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
##       <dbl>      <dbl>      <dbl>      <dbl>    <lgl>
## 1       5.1       3.5       1.4       0.2     NA
## 2       4.9       3.0       1.4       0.2     NA
## 3       4.7       3.2       1.3       0.2     NA
## 4       4.6       3.1       1.5       0.2     NA
## 5       5.0       3.6       1.4       0.2     NA
## 6       5.4       3.9       1.7       0.4     NA
## 7       4.6       3.4       1.4       0.3     NA
## 8       5.0       3.4       1.5       0.2     NA
```

Lectura Archivos de Programas Estadísticos en R

Como ya sabemos, R es un lenguaje de programación y un entorno de computación para computación estadística. Su naturaleza de código fuente abierto ha hecho que en los últimos años prolifere ante alternativas a programas estadísticos de tipo comercial, como SPSS, SAS, etc.

En esta sección, veremos como podemos importar datos desde programas estadísticos avanzados. Así mismo, examinaremos los paquetes que necesitamos instalar para leer nuestros datos en R, de igual manera que hemos hecho con los datos almacenados en archivos de texto o Excel.

Lectura de Archivos SPSS en R

Si somos usuarios del programa SPSS y deseamos importar nuestros archivos SPSS a R, en primer lugar necesitaremos instalar el paquete `haven` que forma parte del ecosistema `tidyverse`.

Instalación

La forma mas fácil de instalar `haven` es instalar el ecosistema `tidyverse` :

```
install.packages("tidyverse")
```

Alternativamente, para instalar únicamente `haven` :

```
install.packages("haven")
```

Por último, podemos instalar la última versión en desarrollo desde su repositorio en GitHub:

```
# install.packages('devtools')
devtools::install_github("tidyverse/haven")
```

Nota que la última opción requiere de la instalación del paquete `devtools` .

Uso

Lo primero que haremos será cargar el ecosistema `tidyverse` :

```
library(tidyverse)
```

También tenemos la opción de cargar únicamente el paquete `haven` :

```
library(haven)
```

Para leer archivos SPSS desde R haremos uso de la función `read_sav()` :

```
# Lectura de los datos SPSS
spss <- read_sav("data/mtcars.sav")
# El objeto de salida es un tibble
spss
## # A tibble: 32 x 11
##       mpg   cyl  disp    hp  drat    wt  qsec    vs    am  gear  carb
##       <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
## 1  21.0     6 160.0   110  3.90  2.620 16.46     0     1     4     4
## 2  21.0     6 160.0   110  3.90  2.875 17.02     0     1     4     4
## 3  22.8     4 108.0    93  3.85  2.320 18.61     1     1     4     1
## 4  21.4     6 258.0   110  3.08  3.215 19.44     1     0     3     1
## 5  18.7     8 360.0   175  3.15  3.440 17.02     0     0     3     2
## 6  18.1     6 225.0   105  2.76  3.460 20.22     1     0     3     1
## 7  14.3     8 360.0   245  3.21  3.570 15.84     0     0     3     4
## 8  24.4     4 146.7    62  3.69  3.190 20.00     1     0     4     2
## 9  22.8     4 140.8    95  3.92  3.150 22.90     1     0     4     2
## 10 19.2     6 167.6   123  3.92  3.440 18.30     1     0     4     4
## # ... with 22 more rows
```

Por supuesto `haven` nos permite gravar nuestros datos en un archivo SPSS con la ayuda de la función `write_sas` :

```
# Escritura del dataframe `mtcars` a un archivo SPSS
write_sav(mtcars, "data/mtcars.sav")
```

Lectura de Archivos Stata en R

Como en el caso anterior utilizaremos el paquete `haven` y pero en este caso utilizaremos la función `read_stata()` :

```
# Lectura de los datos STATA
stata <- read_dta("data/mtcars.dta")
# Mostramos las 6 primeras filas
stata
## # A tibble: 32 x 11
##       mpg   cyl  disp    hp  drat    wt  qsec    vs    am  gear  carb
##       <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
## 1  21.0     6 160.0   110  3.90  2.620 16.46     0     1     4     4
## 2  21.0     6 160.0   110  3.90  2.875 17.02     0     1     4     4
## 3  22.8     4 108.0    93  3.85  2.320 18.61     1     1     4     1
## 4  21.4     6 258.0   110  3.08  3.215 19.44     1     0     3     1
## 5  18.7     8 360.0   175  3.15  3.440 17.02     0     0     3     2
## 6  18.1     6 225.0   105  2.76  3.460 20.22     1     0     3     1
## 7  14.3     8 360.0   245  3.21  3.570 15.84     0     0     3     4
## 8  24.4     4 146.7    62  3.69  3.190 20.00     1     0     4     2
## 9  22.8     4 140.8    95  3.92  3.150 22.90     1     0     4     2
## 10 19.2     6 167.6   123  3.92  3.440 18.30     1     0     4     4
## # ... with 22 more rows
```

De igual manera que en el caso anterior podemos exportar nuestros datos a STATA sin embargo, para archivos Stata utilizaremos la función `write_dta()`:

```
# Escritura del dataframe `mtcars` a un archivo STATA
write_dta(mtcars, "data/mtcars.dta")
```

Lectura de Archivos SAS en R

De la misma forma que en los dos casos anteriores utilizaremos el paquete `haven`, pero en este caso utilizaremos la función `read_sas()` para leer nuestros datos SAS dentro de R:

```
# Lectura de los datos STATA
sas <- read_sas("data/mtcars.sas7bdat")
# Mostramos los datos por pantalla
sas
## # A tibble: 32 x 11
##   mpg   cyl  disp    hp  drat    wt  qsec    vs    am  gear  carb
##   <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
## 1 21.0     6 160.0   110  3.90  2.620 16.46     0     1     4     4
## 2 21.0     6 160.0   110  3.90  2.875 17.02     0     1     4     4
## 3 22.8     4 108.0    93  3.85  2.320 18.61     1     1     4     1
## 4 21.4     6 258.0   110  3.08  3.215 19.44     1     0     3     1
## 5 18.7     8 360.0   175  3.15  3.440 17.02     0     0     3     2
## 6 18.1     6 225.0   105  2.76  3.460 20.22     1     0     3     1
## 7 14.3     8 360.0   245  3.21  3.570 15.84     0     0     3     4
## 8 24.4     4 146.7    62  3.69  3.190 20.00     1     0     4     2
## 9 22.8     4 140.8    95  3.92  3.150 22.90     1     0     4     2
## 10 19.2    6 167.6   123  3.92  3.440 18.30    1     0     4     4
## # ... with 22 more rows
```

De manera semejante podemos exportar nuestros datos a STATA, aunque en esta ocasión utilizaremos la función `write_sas()`:

```
# Escritura del dataframe `mtcars` a un archivo SAS
write_sas(mtcars, "data/mtcars.sas7bdat")
```

Lectura de Archivos Systat en R

Si deseamos importar archivos Systat en R, en esta caso tenemos que hacer uso del paquete `foreign`, como podemos ver a continuación:

```
# Instalamos el paquete `foreign`
install.packages("foreign")
# Activamos la libreria `foreign`
library(foreign)
# Leemos los datos Systat
datos <- read.systat("<ruta archivo>")
```

Lectura de Archivos Minitab en R

De igual manera que en el caso anterior utilizaremos el mismo paquete, pero en este caso utilizaremos la función `read.mtp()`:

```
# Instalamos el paquete `foreign`  
install.packages("foreign")  
# Activamos la libreria `foreign`  
library(foreign)  
# Leemos los datos Systat  
datos <- read.mtp("<ruta archivo>")
```

Lectura de Archivos Jerarquizados

En ocasiones necesitaremos importar datos de sitios web. La mayoría de proveedores proporcionan los datos en formato XML y JSON. En esta sección, aprenderemos como leer datos de archivos JSON y XML.

Además, en el último apartado veremos como importar tablas HTML alojadas en sitios web.

Lectura de archivos JSON en R

Para importar archivos [JSON](#), primero necesitamos instalar y/o cargar el paquete [jsonlite](#).

Instalación

Podemos instalar `jsonlite` desde CRAN:

```
# Instalamos el paquete  
install.packages("jsonlite")
```

Uso

Para importar archivos JSON haremos uso de la función `fromJSON()`:

```
# Cargamos `rjson`  
library(jsonlite)
```

```
# Importamos los datos a un dataframe desde Github en un archivo json
datos <- fromJSON("https://api.github.com/users/rsanchezs/repos")
# Mostramos las variables de `datos`
names(datos)
## [1] "id"                 "name"                "full_name"
## [4] "owner"               "private"              "html_url"
## [7] "description"        "fork"                 "url"
## [10] "forks_url"           "keys_url"              "collaborators_url"
## [13] "teams_url"            "hooks_url"              "issue_events_url"
## [16] "events_url"           "assignees_url"          "branches_url"
## [19] "tags_url"              "blobs_url"              "git_tags_url"
## [22] "git_refs_url"         "trees_url"              "statuses_url"
## [25] "languages_url"        "stargazers_url"          "contributors_url"
## [28] "subscribers_url"      "subscription_url"        "commits_url"
## [31] "git_commits_url"       "comments_url"             "issue_comment_url"
## [34] "contents_url"          "compare_url"             "merges_url"
## [37] "archive_url"            "downloads_url"            "issues_url"
## [40] "pulls_url"              "milestones_url"           "notifications_url"
## [43] "labels_url"              "releases_url"             "deployments_url"
## [46] "created_at"              "updated_at"              "pushed_at"
## [49] "git_url"                  "ssh_url"                 "clone_url"
## [52] "svn_url"                  "homepage"                "size"
## [55] "stargazers_count"        "watchers_count"           "language"
## [58] "has_issues"                "has_projects"             "has_downloads"
## [61] "has_wiki"                  "has_pages"                 "forks_count"
## [64] "mirror_url"                "archived"                  "open_issues_count"
## [67] "forks"                      "open_issues"                "watchers"
## [70] "default_branch"
```

```
# Accedemos a los nombres de mis repositorios
datos$name
## [1] "appliedStatsR"
## [2] "ciencia-de-datos-con-r"
## [3] "ciencia-de-datos-con-r-casos-de-estudio"
## [4] "ciencia-de-datos-con-r-libro"
## [5] "courses"
## [6] "datasciencecoursera"
## [7] "datasharing"
## [8] "dplyr"
## [9] "EHairdressing"
## [10] "jekyll-now"
## [11] "manipulacion-datos-con-r"
## [12] "MITx_6_00_1x"
## [13] "probability"
## [14] "programacion-en-r"
## [15] "programacion-estadistica-r"
## [16] "PY4E"
## [17] "r4ds"
## [18] "RGraphic"
## [19] "rprogramming"
## [20] "rsanchezs.github.io"
## [21] "statsR"
## [22] "videoRprogramming"
## [23] "visualizacion-de-datos-con-r"
## [24] "webinars"
```

Lectura de archivos XML en R

Si deseamos importar archivos [XML](#), una de los métodos más fáciles es mediante el uso del paquete [XML2](#).

Instalación

Podemos instalar `xml2` desde CRAN:

```
# Instalación de `xml2` desde CRAN
install.packages("xml2")
```

o podemos instalar la versión en desarrollo en GitHub, con la ayuda de `devtools`:

```
# Instalación desde el repositorio de GitHub
install_github("r-lib/xml2")
```

Uso

Mediante la función `read_xml()` podemos importar archivos XML:

```
# Cargamos el paquete
library(xml2)
# Importamos los datos XML
archivo_xml <- read_xml("data/empleados.xml")
archivo_xml
## {xml_document}
## <PLANTILLA>
## [1] <EMPLEADO>\n  <ID>1</ID>\n  <NOMBRE>Ruben</NOMBRE>\n  <SALARIO>623.3 ...
## [2] <EMPLEADO>\n  <ID>2</ID>\n  <NOMBRE>Ramon</NOMBRE>\n  <SALARIO>515.2 ...
## [3] <EMPLEADO>\n  <ID>3</ID>\n  <NOMBRE>Tomas</NOMBRE>\n  <SALARIO>611<...
## [4] <EMPLEADO>\n  <ID>4</ID>\n  <NOMBRE>Marga</NOMBRE>\n  <SALARIO>729<...
## [5] <EMPLEADO>\n  <ID>5</ID>\n  <NOMBRE>Miguel</NOMBRE>\n  <SALARIO>843. ...
## [6] <EMPLEADO>\n  <ID>6</ID>\n  <NOMBRE>Nuria</NOMBRE>\n  <SALARIO>578<...
## [7] <EMPLEADO>\n  <ID>7</ID>\n  <NOMBRE>Jaime</NOMBRE>\n  <SALARIO>632.8 ...
## [8] <EMPLEADO>\n  <ID>8</ID>\n  <NOMBRE>Dani</NOMBRE>\n  <SALARIO>722.5<...
```

En primer lugar, podríamos estar interesados en el número de nodos que contienen el archivo y que podemos conocer como se muestra a continuación:

```
# Mostramos el número de nodos del archivo
xml_length(archivo_xml)
## [1] 8
```

Podemos acceder a un nodo con la ayuda de la función `xml_child()` como se muestra en el siguiente fragmento de código:

```
# Accedemos al primer nodo
xml_child(archivo_xml, 1)
## {xml_node}
## <EMPLEADO>
## [1] <ID>1</ID>
## [2] <NOMBRE>Ruben</NOMBRE>
## [3] <SALARIO>623.3</SALARIO>
## [4] <ALTA>1/1/2012</ALTA>
## [5] <DEPT>IT</DEPT>
```

Para conocer todos los valores de un nodo que coinciden con una expresión `xpath` podemos hacerlo como se muestra a continuación:

```
# Conocer todos los valores que coinciden con un elemento
nombre <- xml_find_all(archivo_xml, "./NOMBRE")
nombre
## {xml_nodeset (8)}
## [1] <NOMBRE>Ruben</NOMBRE>
## [2] <NOMBRE>Ramon</NOMBRE>
## [3] <NOMBRE>Tomas</NOMBRE>
## [4] <NOMBRE>Marga</NOMBRE>
## [5] <NOMBRE>Miguel</NOMBRE>
## [6] <NOMBRE>Nuria</NOMBRE>
## [7] <NOMBRE>Jaime</NOMBRE>
## [8] <NOMBRE>Dani</NOMBRE>
```

Con la ayuda de la función `xml_text()`, `xml_double` y `xml_integer` obtendremos un vector de caracteres, reales o enteros respectivamente del documento, nodo o conjunto de nodos:

```
# Convertir un documento, nodo o conjunto de nodo a vectores
nombres <- xml_text(nombre)
nombres
## [1] "Ruben"  "Ramon"  "Tomas"  "Marga"  "Miguel" "Nuria"  "Jaime"  "Dani"
```

Con lo visto hasta ahora podemos pasar el documento XML a un dataframe para nuestro análisis como se muestra a continuación:

```
# Obtenemos los valores de cada nodo en un vector
id <- xml_integer(xml_find_all(archivo_xml, "./ID"))
nombre <- xml_text(xml_find_all(archivo_xml, "./NOMBRE"))
salario <- xml_double(xml_find_all(archivo_xml, "./SALARIO"))
alta <- xml_text(xml_find_all(archivo_xml, "./ALTA"))
dept <- xml_text(xml_find_all(archivo_xml, "./DEPT"))

# Creamos un dataframe a partir de los vectores
plantilla <- data.frame(id, nombre, salario, alta, dept)
plantilla
##   id nombre salario      alta      dept
## 1  1 Ruben  623.30  1/1/2012       IT
## 2  2 Ramon  515.20  9/23/2013 Produccion
## 3  3 Tomas  611.00 11/15/2014       IT
## 4  4 Marga  729.00  5/11/2014       HR
## 5  5 Miguel 843.25  3/27/2015  Finanzas
## 6  6 Nuria  578.00  5/21/2013  Limpieza
## 7  7 Jaime  632.80  7/30/2013     I+D
## 8  8 Dani   722.50  6/17/2014 Produccion
```

Lectura de tablas HTML en R

Para importar tablas HTML necesitaremos del paquete `rvest`.

Instalación

Para conseguir la última versión desde CRAN

```
# Instalación de `rvest` desde CRAN
install.packages("rvest")
```

Para descargar la versión en desarrollo desde su repositorio en GitHub:

```
# install.packages('devtools')
devtools::install_github("hadley/rvest")
```

Uso

Cargamos la libreria `rvest`:

```
# Cargamos la libreria `rvest`
library(rvest)
```

Para el ejemplo, además haremos uso del paquete `xml2` para descargar una tabla de la siguiente [página](#) con los nombres de niño y niña más comunes en 2017:

```
# Cargamos la libreria `xml2`
library(xml2)
```

Descargamos la página mediante `xml2::read_html()` y con la ayuda de `xml2::xml_find_first()` encontramos el nodo que coincide con `xpath = "//table"`:

```
# Descargamos la página
html <- read_html("http://www.enterat.com/servicios/nombres-nino-nina.php")
# Encontramos el elemento que coincide con la clase `table`
tabla <- xml_find_first(html, xpath = "//table")
```

Por último, mediante `rvest::html_table` analizamos sintácticamente la tabla html y la transformamos en un `dataframe`:

```
# Pasamos la tabla HTML a un dataframe
df <- html_table(tabla)
head(df)

##          X1           X2           X3
## 1 Posici n Nombres de ni o Nombres de ni a
## 2      1        HUGO        LUCIA
## 3      2        DANIEL     MARTINA
## 4      3        MARTIN     MARIA
## 5      4        PABLO      SOFIA
## 6      5 ALEJANDRO    PAULA
```

Lectura de Bases de Datos

Una fuente común dónde podemos encontrar datos es en las bases de datos relacionales. Actualmente existen multitud de Sistemas Gestores de Bases de datos (SGBD) para trabajar con bases de datos relacionales, y R puede conectarse a la gran mayoría de los mismos. El paquete `DBI` proporciona una sintaxis para el acceso a varios SGBD. Actualmente soporta SQLite, MySQL, MariaDB, PostgreSQL, y Oracle, además de un **wrapper** (del inglés, envoltorio) para el API Java DataBase Connectivity (JBDC).

Para conectarnos en una base de datos, necesitamos dos paquetes:

- El paquete `DBI`, que consiste en la definición de una interface para la comunicación entre R y SGBDR.
- El paquete `R<nombre_SGBDR>` que consiste en la implementación del driver R/SGBDR.

Conexión a SQLite

Para conectarnos a una base de datos SQLite, en primer lugar debemos instalar y cargar los paquetes `DBI` y `RSQlite`

```
# Instalamos el paquetes
install.packages(c("DBI", "RSQlite"))
```

```
# Cargamos los paquetes
library(DBI)
library(RSQLite)
## Error in library(RSQLite): there is no package called 'RSQLite'
```

Después definimos la conexión a la base de datos, en la que especificaremos el driver para que sea de tipo "SQLite" y que pasaremos a la función `dbConnect()` a la que además le pasaremos como argumento la ruta del archivo de la base de datos

```
# Definimos el driver
driver <- dbDriver("SQLite")
## Error: Couldn't find driver SQLite. Looked in:
## * global namespace
## * in package called SQLite
## * in package called RSSQLite
# Definimos la ruta al archivo de la bd
archivo_bd <- system.file("data/database.sqlite")
# Establecemos la conexión
con <- dbConnect(driver, archivo_bd)
## Error in dbConnect(driver, archivo_bd): object 'driver' not found
```

Conexión a MySQL

La única diferencia para la conexión a una base de datos MySQL es cargar el paquete `RMySQL`, establecer el driver a "MySQL" y por último, proporcionar el usuario y contraseña:

```
# Instalamos el paquetes
install.packages(c("DBI", "RMySQL"))
# Cargamos los paquetes
library(DBI)
library(RMySQL)
```



```
## Error in library(RMySQL): there is no package called 'RMySQL'
```

```
# Definimos el driver
driver <- dbDriver("MySQL")
# Establecemos la conexión
conn <- dbConnect(driver, user = "ruben", password = "1234", db = "sakila")
```

Conexión a PostgreSQL, Oracle y JDBC

Para establecer la conexión a PostgreSQL, Oracle y JDBC procederíamos como en el apartado anterior, pero en estos casos requieren de los paquetes `RPostgreSQL`, `ROracle` y `RJDBC` respectivamente.

Listar y recuperar datos

Para listar las tablas de una base de datos haremos uso de la función `dbListTables()`:

```
## Error in dbWriteTable(con, "mtcars", mtcars): object 'con' not found
```

```
# Mostramos las tablas de la bd
dbListTables(con)
## Error in dbListTables(con): object 'con' not found
```

Por otro lado, para recuperar los datos de una base de datos escribiremos una consulta, que consiste en un string que contiene una instrucción SQL, y la enviaremos a la base de datos con la ayuda de la función `dbGetQuery`. En el siguiente ejemplo, `SELECT * FROM mtcars` significa "muestra cada columna de la tabla con nombre `mtcars`":

```
# Creamos una consulta
consulta <- "SELECT * FROM mtcars"
# Enviamos la consulta al SGBDR
datos <- dbGetQuery(con, consulta)
## Error in dbGetQuery(con, consulta): object 'con' not found
# Mostramos por pantalla un subconjunto de los datos
datos[1:10, 1:5]
## Error in eval(expr, envir, enclos): object 'datos' not found
```

Para aquellos que no conocen el lenguaje SQL, el paquete `DBI` proporciona multitud de funciones para la manipulación de base de datos. A modo de ejemplo, mediante la función `dbReadTable` conseguiremos el mismo resultado que en código anterior:

```
# Consultamos la tabla `mtcars`
dbReadTable(con, "mtcars")
```

Desconexión de la base de datos

Por último, una vez hemos manipulado la base de datos, es necesario desconectarse y descargar el driver:

```
# Cerramos la conexión
dbDisconnect(con)
## Error in dbDisconnect(con): object 'con' not found
```