

# Aprender R: iniciación y perfeccionamiento

*François Rebaudo*

*2018-11-19*



# Contents

<b>1</b>	<b>Preámbulo</b>	<b>5</b>
<b>2</b>	<b>Agradecimientos</b>	<b>7</b>
<b>3</b>	<b>Licencia</b>	<b>9</b>
<b>4</b>	<b>Introducción</b>	<b>11</b>
4.1	¿Por qué aprender R?	11
4.2	Este libro	11
4.3	Lectura adicional en español	11
<b>I</b>	<b>Conceptos básicos</b>	<b>13</b>
<b>5</b>	<b>Primeros pasos</b>	<b>15</b>
5.1	Instalar R	15
5.2	R como calculadora	15
5.3	El concepto de objeto	22
5.4	Los scripts	24
5.5	Conclusión	25
<b>6</b>	<b>Elegir un entorno de desarrollo</b>	<b>27</b>
6.1	Editores de texto y entorno de desarrollo	27
6.2	RStudio	27
6.3	Notepad++ avec Npp2R	29
6.4	Geany	33
6.5	Otras soluciones	34
6.6	Conclusión	34
<b>7</b>	<b>Tipos de datos</b>	<b>37</b>
7.1	El tipo <code>numeric</code>	37
7.2	El tipo <code>character</code>	39
7.3	El tipo <code>factor</code>	41
7.4	El tipo <code>logical</code>	42
7.5	Acerca de <code>NA</code>	42
7.6	Conclusión	43
<b>8</b>	<b>Contenedores de datos</b>	<b>45</b>
8.1	El contenedor <code>vector</code>	45
8.2	El contenedor <code>list</code>	52
8.3	El contenedor <code>data.frame</code>	64
8.4	El contenedor <code>matrix</code>	68
8.5	El contenedor <code>array</code>	74
8.6	Conclusión	76

<b>9 Las funciones</b>	<b>77</b>
9.1 ¿Qué es una función?	77
9.2 Las funciones más comunes	77
9.3 Otras funciones útiles	94
9.4 Algunos ejercicios para practicar	98
9.5 Escribir una función	100
9.6 Otras funciones desarrolladas por la comunidad de usuarios: los <i>packages</i>	104
9.7 Conclusión	106
<b>10 Importar y exportar datos</b>	<b>107</b>
10.1 Leer datos de un archivo	107
10.2 Guardar datos para R	110
10.3 Exportar datos	111
10.4 Conclusión	112
<b>11 Algorítmico</b>	<b>113</b>
11.1 Pruebas lógicas con <i>if</i>	113
11.2 Pruebas lógicas con <i>switch</i>	116
11.3 El bucle <i>for</i>	117
11.4 El bucle <i>while</i>	121
11.5 El bucle <i>repeat</i>	122
11.6 <i>next</i> y <i>break</i>	123
11.7 Los bucles de la familia <i>apply</i>	125
11.8 Conclusión	132
<b>12 Gestión de proyectos con R</b>	<b>133</b>
12.1 Gestionando archivos y directorios de trabajo.	133
12.2 Gestión de versiones de script	133
12.3 Gestion de documentacion	134
12.4 Conclusión	136
<b>II Estudio de caso</b>	<b>137</b>
<b>13 Analizar datos de loggers de temperatura</b>	<b>139</b>

# Chapter 1

## Preámbulo

Este libro está incompleto por el momento y está viendo su versión preliminar. Muchos capítulos ya están en línea, así que no dude en consultarlos y comenzar su iniciación o su perfeccionamiento en el lenguaje de programación R.

Si tiene algún comentario, sugerencia o si identifica errores, no dude en enviarme un correo electrónico ([francois.rebaudo@ird.fr](mailto:francois.rebaudo@ird.fr)<sup>1</sup>), o si está familiarizado con GitHub en el sitio web del proyecto ([https://github.com/frareb/myRBook\\_SP](https://github.com/frareb/myRBook_SP)). Este libro es colaborativo, se basa en su participación.

Este libro también está disponible en francés (<http://myrbookfr.netlify.com/>)

Últimas modificaciones:

### 09/11/2018

- Capítulo Gestión de proyectos con R
- La primera parte de este libro sobre conceptos básicos está completa

### 08/11/2018

- algorítmico (5/6) `next` y `break`
- algorítmico (6/6) Los bucles de la familia `apply`

### 19/10/2018

- algorítmico (3/6) el bucle `for`
- algorítmico (4/6) el bucle `while`
- algorítmico (4'/6) el bucle `repeat`

### 28/09/2018

- algorítmico (1/6) Pruebas lógicas con `if`
- algorítmico (2/6) Pruebas lógicas con `switch`

### 17/09/2018

- Introducción

### 10/09/2018

- Importar y exportar datos (parte 2-3/3): Guardar y exportar datos

### 06/09/2018

- Importar y exportar datos (parte 1/3): Leer datos desde un archivo

### 30/08/2018

---

<sup>1</sup><mailto:francois.rebaudo@ird.fr>

- Nuevo estudio de caso: dataloggers
- Modificaciones de Marc G. sobre el tipo de datos `numeric` y los entornos de desarrollo

**24/08/2018**

- Capítulo sobre las funciones (3/3)

**27/07/2018**

- Capítulo sobre las funciones (2/3)

**25/07/2018**

- Capítulo sobre las funciones (1/3)

**17/07/2018**

- edición y corrección del español (Susi L.)
- tercera parte del capítulo *Contenedores de datos*: El contenedor *data.frame*
- cuarta parte del capítulo *Contenedores de datos*: El contenedor *matrix*
- quinta parte del capítulo *Contenedores de datos*: El contenedor *array*

**16/07/2018**

- edición y corrección del español (Estefanía Q.)

**12/07/2018**

- segunda parte del capítulo *Contenedores de datos*: El contenedor *list*

**06/07/2018**

- edición y corrección del español (Camila B.)
- primera parte del capítulo *Contenedores de datos*: El contenedor *vector*

**04/07/2018**

- tabla de contenidos con los próximos capítulos
- error de tipografía en *Elegir un entorno de desarrollo*

**02/07/2018**

- tres capítulos en línea (*primeros pasos*, *elegir un entorno de desarrollo*, *tipos de datos*)

## Chapter 2

# Agradecimientos

Agradezco a todos los colaboradores que ayudaron a mejorar este libro con sus consejos, sugerencias de cambios y correcciones (en orden alfabético):

```
## Colaboradores :  
Camila Benavides Frias (Bolivia)  
Marc Girondot (France ; UMR 8079 ESE)  
Susi Loza Herrera (Bolivia)  
Estefania Quenta Herrera (Bolivia)
```

Las versiones de gitbook, html y epub de este libro usan los iconos de fuente abierta de Font Awesome (<https://fontawesome.com>). La versión en PDF utiliza los iconos del proyecto Tango disponibles en openclipart (<https://openclipart.org/>). Este libro fue escrito con el paquete R bookdown (<https://bookdown.org/>). El código fuente está disponible en GitHub ([https://github.com/frareb/myRBook\\_SP](https://github.com/frareb/myRBook_SP)). La versión en línea se aloja y actualiza a través de Netlify (<http://netlify.com/>).





## Chapter 3

# Licencia

Licencia Reconocimiento-NoComercial-SinObraDerivada 3.0 España (CC BY-NC-ND 3.0 ES ; <https://creativecommons.org/licenses/by-nc-nd/3.0/es/>)

Esto es un resumen inteligible para humanos (y no un sustituto) de la licencia.

### **Usted es libre de:**

- Compartir — copiar y redistribuir el material en cualquier medio o formato.
- El licenciador no puede revocar estas libertades mientras cumpla con los términos de la licencia.

### **Bajo las condiciones siguientes:**

- Reconocimiento — Debe reconocer adecuadamente la autoría, proporcionar un enlace a la licencia e indicar si se han realizado cambios. Puede hacerlo de cualquier manera razonable, pero no de una manera que sugiera que tiene el apoyo del licenciador o lo recibe por el uso que hace.
- NoComercial — No puede utilizar el material para una finalidad comercial.
- SinObraDerivada — Si remezcla, transforma o crea a partir del material, no puede difundir el material modificado.
- No hay restricciones adicionales — No puede aplicar términos legales o medidas tecnológicas que legalmente restrinjan realizar aquello que la licencia permite.

### **Avisos:**

No tiene que cumplir con la licencia para aquellos elementos del material en el dominio público o cuando su utilización esté permitida por la aplicación de una excepción o un límite. No se dan garantías. La licencia puede no ofrecer todos los permisos necesarios para la utilización prevista. Por ejemplo, otros derechos como los de publicidad, privacidad, o los derechos morales pueden limitar el uso del material.



## Chapter 4

# Introducción

### 4.1 ¿Por qué aprender R?

Porque R se ha convertido en una herramienta esencial para el análisis y la gestión de datos científicos, y en este contexto se vuelve esencial dominar al menos los conceptos básicos. El éxito de R no es una coincidencia: R es un software que todos pueden obtener libremente garantizando la transparencia y la reproducibilidad de los resultados científicos (sujeto a cumplir con algunas reglas que abordará este libro). R también se basa en una comunidad muy activa con varios miles de módulos adicionales (paquetes) para realizar el análisis estadístico más avanzado.

### 4.2 Este libro

El propósito de este libro es proporcionar a los estudiantes y aquellos que deseen aprender sobre R una base sólida para luego implementar sus propios proyectos científicos y la valoración de sus resultados. Hay muchos libros dedicados a R, pero ninguno cubre los elementos básicos de este lenguaje con el fin de hacer que los resultados científicos sean publicables y reproducibles.

En general, este libro está dirigido a toda la comunidad científica y en particular a aquellos interesados en las ciencias de la vida, y los ejemplos en este libro se basarán en estudios de biología.

Este libro nació de la solicitud de los estudiantes de las universidades que colaboran con el IRD en América del Sur. Por lo tanto, su primera versión está escrita en español (hay pocos documentos de calidad en R en español). Comencé su traducción al francés en 2018 y hoy ambas versiones coevolucionan con contenido que puede variar (por ejemplo, para estudios de casos).

### 4.3 Lectura adicional en español

- R para Principiantes, Emmanuel Paradis ([https://cran.r-project.org/doc/contrib/rdebuts\\_es.pdf](https://cran.r-project.org/doc/contrib/rdebuts_es.pdf))



## **Part I**

# **Conceptos básicos**



## Chapter 5

# Primeros pasos

### 5.1 Instalar R

El programa para instalar el software R se puede descargar desde el sitio web de R: <https://www.r-project.org/>. En el sitio web de R, primero es necesario elegir un espejo CRAN (servidor desde el cual se debe descargar R, y desde el más cercano a su ubicación geográfica), luego descargue el archivo *base*. Los usuarios de Linux pueden preferir un `sudo apt-get install r-base`.

El software R se puede descargar de muchos servidores CRAN (Comprehensive R Archive Network) de todo el mundo. Estos servidores se llaman espejos. La elección del espejo es manual.

### 5.2 R como calculadora

Una vez que se inicia el programa, aparece una ventana cuya apariencia puede variar dependiendo de su sistema operativo (Figura 5.1). Esta ventana se llama *console*.

La consola corresponde a la interfaz donde se interpretará el código, es decir, donde el código será transformado en lenguaje de máquina, ejecutado por la computadora y retransmitido en forma legible por humanos. Esto es análogo a lo que sucede en una calculadora (Figura 5.2). Así es como se usará R más adelante en esta sección.

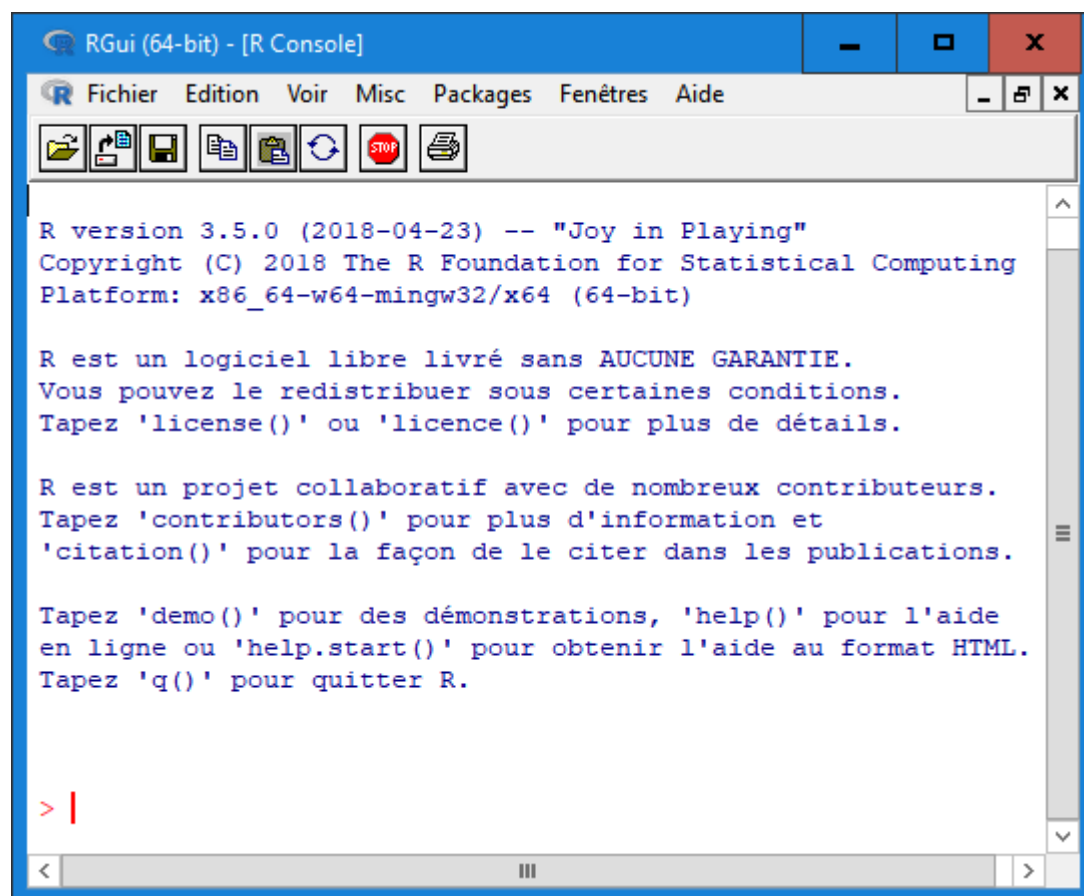
A lo largo de este libro, los ejemplos del código R aparecerán sobre un fondo gris. Se pueden copiar y pegar directamente en la consola, aunque es mejor reproducir los ejemplos escribiéndolos en la consola (o más adelante en los scripts) para una mejor comprensión del manejo del programa R. El resultado de lo que se envía en la consola también aparecerá en un fondo gris con `##` delante del código para hacer la distinción entre el código y el resultado del código.

#### 5.2.1 Los operadores aritméticos

```
5 + 5
```

```
## [1] 10
```

Si escribimos `5 + 5` en la consola y luego `Enter`, el resultado aparece precedido por el número `[1]` entre corchetes. Este número corresponde al número del resultado (en nuestro caso, solo hay un resultado, volveremos a este aspecto más adelante). También podemos observar en este ejemplo el uso de espacios antes y después del signo `+`. Estos espacios no son necesarios, pero permiten que el código sea más legible para los humanos (es decir, más agradable de leer tanto para nosotros como para



**Figure 5.1:** Captura de pantalla de la consola R en Windows.

**Table 5.1:** Operadores aritméticos.

Label	Operador
adición	+
resta	-
multiplicación	*
división	/
potencia	^
módulo	%%
cociente decimal	%/%

las personas con las que queremos compartir nuestro código). Los operadores aritméticos disponibles en R se resumen en la tabla 5.1.

Clásicamente, las multiplicaciones y divisiones tienen prioridad sobre las adiciones y sustracciones. Si es necesario, podemos usar paréntesis.

```
5 + 5 * 2
```

```
## [1] 15
```





**Figure 5.2:** Captura de pantalla de la consola R al lado de la calculadora de Windows.

```
(5 + 5) * 2
```

```
## [1] 20
```

```
(5 + 5) * (2 + 2)
```

```
## [1] 40
```

```
(5 + 5) * ((2 + 2) / 3)^2
```

```
## [1] 17.77778
```

El operador *módulo* corresponde al resto de la división euclidiana. Se usa en ciencias de la computación, por ejemplo, para saber si un número es par o impar (un número módulo 2 devolverá 1 si es impar y 0 si es par).

```
451 %% 2
```

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

```
288 %% 2
```

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

**Table 5.2:** Operadores de comparación.

Label	Operador
más pequeño que	<
mayor que	>
más pequeño o igual a	<=
más grande o igual a	>=
igual a	==
diferente de	!=

```
(5 + 5 * 2) %% 2
```

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

```
((5 + 5) * 2) %% 2
```

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

R también incorpora algunas constantes que incluyen  $\pi$ . Además, el signo infinito está representado por `Inf`.

```
pi
```

```
## [1] 3.141593
```

```
pi * 5^2
```

```
## [1] 78.53982
```

```
1/0
```

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

El *estilo* del código es importante porque el código está destinado a ser leído por nosotros y por otras personas. Para tener un estilo legible, se recomienda colocar espacios antes y después de los operadores aritméticos, excepto “\*”, “/” y “^”, aunque a veces es útil agregarlos como es el caso en nuestros ejemplos.

### 5.2.2 Los operadores comparativos

Sin embargo, R es mucho más que una simple calculadora porque permite otro tipo de operadores: operadores de comparación, para *comparar* los valores (Table 5.2).

Por ejemplo, si queremos saber si un número es más grande que otro, podemos escribir:

```
5 > 3
```

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

R devuelve `TRUE` si la comparación es verdadera y `FALSE` si la comparación es falsa.

```
5 > 3
```

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

```
2 < 1.5
```

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

```
2 <= 2
```

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

```
3.2 >= 1.5
```

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

Podemos combinar operadores aritméticos con operadores de comparación.

```
(5 + 8) > (3 * 45/2)
```

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

En la comparación  $(5 + 8) > (3 * 45/2)$  no se necesitan paréntesis, pero permiten que el código sea más fácil de leer.

Un operador de comparación particular es *igual a*. Veremos en la siguiente sección que el signo = está reservado para otro uso: permite asignar un valor a un objeto. El operador de comparación *igual a* debe ser diferente, por eso R usa ==.

```
42 == 53
```

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

```
58 == 58
```

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

Otro operador particular es *diferente de*. Se usa con *un signo de admiración* seguido de *igual*, !=. Este operador permite obtener la respuesta opuesta a ==.

```
42 == 53
```

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

```
42 != 53
```

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

```
(3 + 2) != 5
```

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

```
10/2 == 5
```

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

R usa TRUE y FALSE, que también son valores que se pueden probar con operadores de comparación. Pero R también asigna un valor a TRUE y FALSE:

```
TRUE == TRUE
```

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

```
TRUE > FALSE
```

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

```
1 == TRUE
```

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

```
0 == FALSE
```

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

```
TRUE + 1
```

```
## [1] 2
```

```
FALSE + 1
```

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

```
(FALSE + 1) == TRUE
```

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

El valor de TRUE es 1 y el valor de FALSE es 0. Veremos más adelante cómo usar esta información en los próximos capítulos.

R es también un lenguaje relativamente permisivo, significa que admite cierta flexibilidad en la forma de escribir el código. Debatir sobre la idoneidad de esta flexibilidad está fuera del alcance de este libro, pero podemos encontrar en el código R en Internet o en otras obras el atajo T para TRUE y F for FALSE.

```
T == TRUE
```

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

```
F == FALSE
```

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

```
T == 1
```

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

```
F == 0
```

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

```
(F + 1) == TRUE
```

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

**Table 5.3:** Operadores lógicos.

Label	Operador
no es	!
y	&
o	
o exclusivo	xor()

Aunque esta forma de referirse a TRUE y FALSE por T y F está bastante extendida, en este libro siempre usaremos TRUE y FALSE para que el código sea más fácil de leer. Como mencionado anteriormente, el objetivo de un código no solo es ser funcional sino también fácil de leer y volver a leer.

### 5.2.3 Los operadores lógicos

Hay un último tipo de operador, los operadores lógicos. Estos son útiles para combinar operadores de comparación (Table 5.3).

```
!TRUE
```

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

```
!FALSE
```

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

```
((3 + 2) == 5) & ((3 + 3) == 5)
```

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

```
((3 + 2) == 5) & ((3 + 3) == 6)
```

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

```
(3 < 5) & (5 < 5)
```

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

```
(3 < 5) & (5 <= 5)
```

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

El operador lógico `xor()` es *o exclusivo*. Es decir, uno de los dos **argumentos** de la **función** `xor()` debe ser verdadero, pero no ambos. Más adelante volveremos a las **funciones** y sus **argumentos**, pero recuerde que identificamos una función por sus paréntesis que contienen argumentos separados por comas.

```
xor((3 + 2) == 5, (3 + 3) == 6)
```

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

```
xor((3 + 2) == 5, (3 + 2) == 6)
```

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

```
xor((3 + 3) == 5, (3 + 2) == 6)
```

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

```
xor((3 + 3) == 5, (3 + 3) == 6)
```

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

Se recomienda que las comas , sean seguidas de un espacio para que el código sea más agradable de leer.

### 5.2.4 Ayuda a los operadores

El archivo de ayuda en inglés sobre operadores aritméticos se puede obtener con el comando `? '+'`. El de los operadores de comparación con el comando `? '=='` y el de los operadores lógicos con el comando `? '&'`.

## 5.3 El concepto de objeto

Un aspecto importante de la programación con R, pero también la programación en general es la noción de objeto. Como se indica en la página web de wikipedia ([https://ia.wikipedia.org/wiki/Objeto\\_\(informatica\)](https://ia.wikipedia.org/wiki/Objeto_(informatica))), en ciencias de la computación, un objeto es un *contenedor*, es decir, algo que contendrá información. La información contenida en un objeto puede ser muy diversa, pero por el momento contendremos en un objeto el número 5. Para hacer esto (y para reutilizarlo más adelante), debemos darle un nombre a nuestro objeto. En R, los nombres de los objetos no deben contener caracteres especiales como `^ $ ? | + () [] {}`, entre otros. No deben comenzar con un número ni contener espacios. El nombre del objeto debe ser representativo de lo que contiene, sin ser demasiado corto ni demasiado largo. Imagine que nuestro número 5 corresponde al número de repeticiones de un experimento. Nos gustaría darle un nombre que se refiera a *numero* y *repeticiones*, que podríamos reducir a *nbr* y *rep*, respectivamente (*nbr* para number en ingles). Hay varias posibilidades que son bastante comunes bajo R:

- la separación mediante *guión bajo* (underscore): `nbr_rep`
- la separación mediante el carácter *punto*: `nbr.rep`
- el uso de letras minúsculas: `nbrrep`
- el estilo *lowerCamelCase* que consiste en una primera palabra en minúscula y la primera letra de las siguientes palabras con una letra mayúscula: `nbrRep`
- el estilo *UpperCamelCase* donde cada palabra comienza con una letra mayúscula: `NbrRep`

Todas estas formas de nombrar un objeto son equivalentes. En este libro usaremos el estilo *lowerCamelCase*. En general, debemos evitar los nombres que son demasiado largos, como `miNumeroDeRepeticionesDeMiExperimento` o demasiado cortos como `nR`, y los nombres que no permiten identificar los contenidos como `miVariable` o `miNumero`, así que nombres como `a` o `b`. El objetivo es de tener una idea de lo que hay en cada objeto en base a su nombre.

Hay diferentes maneras de definir un nombre para los objetos que crearemos con R. En este libro, utilizamos el estilo *lowerCamelCase*. Lo importante no es la elección del estilo, sino la consistencia en su elección. El objetivo es tener un código funcional, pero también un código que sea fácil y agradable de leer para nosotros y para los demás.

Ahora que hemos elegido un nombre para nuestro objeto, debemos crearlo y hacer que R entienda que nuestro objeto debe contener el número 5. Hay tres maneras de crear un objeto bajo R:

- con `<-`
- con `=`
- o con `->`

```
nbrRep <- 5
nbrRep = 5
5 -> nbrRep
```

En este libro siempre usaremos la forma `<-` para coherencia y también porque es la forma más común.

```
nbrRep <- 5
```

Acabamos de crear un objeto `nbrRep` y establecerlo con el valor 5. Este objeto ahora está disponible en nuestro entorno de computación y puede ser utilizado. Algunos ejemplos :

```
nbrRep + 2
```

```
## [1] 7
```

```
nbrRep * 5 - 45/56
```

```
## [1] 24.19643
```

```
pi * nbrRep^2
```

```
## [1] 78.53982
```

El valor asociado con nuestro objeto `nbrRep` se puede modificar de la misma manera que cuando se creó:

```
nbrRep <- 5
nbrRep + 2
```

```
## [1] 7
```

```
nbrRep <- 10
nbrRep + 2
```

```
## [1] 12
```

```
nbrRep <- 5 * 2 + 7/3
nbrRep + 2
```

```
## [1] 14.33333
```

El uso de objetos tiene sentido cuando tenemos operaciones complejas para realizar y hace que el código sea más agradable de leer y entender.

```
(5 + 9^2 - 1/18) / (32 * 45/8 + 3)
```

```
## [1] 0.4696418
```

```
termino01 <- 5 + 9^2 - 1/18
termino02 <- 32 * 45/8 + 3
termino01 / termino02
```

```
## [1] 0.4696418
```

## 5.4 Los scripts

R es un lenguaje de programación denominado *lenguaje de scripting*. Esto se refiere al hecho de que la mayoría de los usuarios escribirán pequeñas piezas de código en lugar de programas completos. R se puede usar como una simple calculadora, y en este caso no será necesario mantener un historial de las operaciones que se han realizado. Pero si las operaciones a implementar son largas y complejas, puede ser necesario e interesante guardar lo que se ha hecho para poder continuar más adelante. El archivo en el que se almacenarán las operaciones es lo que comúnmente se llama el *script*. Un *script*, por lo tanto, es un archivo que contiene una sucesión de información comprensible por R y que es posible ejecutar.

### 5.4.1 Crear un script y documentarlo

Para crear un nuevo script basta con abrir un documento vacío de texto, que será editado por un editor de texto como el bloc de notas en Windows o Mac OSX, o Gedit o incluso nano en Linux. Por convención, este archivo toma la extensión “.r” o “.R” (lo mas comun). Esta última convención se usará en este libro (“*miArchivo.R*”). Desde la interfaz gráfica de R, es posible crear un nuevo script en Mac OS y Windows a través de *file*, luego *new script* y *save as*. Al igual que el nombre de los objetos, el nombre del script es importante para que podamos identificar fácilmente su contenido. Por ejemplo, podríamos crear un archivo *formRConceptsBase.R* que contenga los objetos que acabamos de crear y los cálculos que hicimos. Pero incluso con nombres de objetos y archivos bien definidos, será difícil recordar el significado de este archivo sin la documentación que acompaña a este script. Para documentar un script utilizaremos *comentarios*. Los *comentarios* son elementos que R identificará como tales y no se ejecutarán. Para especificar a R que vamos a hacer un *comentario*, debemos usar el carácter octothorpe (corsé o numeral) #. Los comentarios se pueden insertar en una nueva línea o al final de la línea.

```
# creación objeto número de repeticiones
nbrRep <- 5 # Comentario de fin de línea
```

Todo lo que hay después del símbolo numeral # no será ejecutado. Significa que podríamos usar comentarios como ### o #comentario, aun que se recomienda hacer comentarios con un solo símbolo numeral seguido por un espacio y después su comentario: # mi comentario.

Los comentarios también se pueden usar para hacer que una línea ya no se ejecute. En este caso no queremos ejecutar la segunda línea:

```
nbrRep <- 5
# nbrRep + 5
```

Para volver a la documentación del script, se recomienda comenzar cada uno de nuestros scripts con una breve descripción de su contenido, luego cuando el script sea extenso, estructurarlo en diferentes partes para facilitar su lectura.

```
# -----
# Aquí hay un script para adquirir los conceptos básicos
# con R
# fecha de creación : 27/06/2018
# autor : François Rebaudo
# -----

# [1] creación del objeto número de repeticiones
# -----

nbrRep <- 5

# [2] cálculos simples
# -----
```



```
pi * nbrRep^2
```

```
## [1] 78.53982
```

Para ir más allá en el estilo del código, una guía completa de recomendaciones está disponible en línea en el sitio web *tidyverse* (en inglés ; <http://style.tidyverse.org/>).

### 5.4.2 Ejecutar un script

Como tenemos un script, no trabajamos directamente en la consola. Pero solo la consola puede *entender* el código R y devolvernos los resultados que queremos obtener. Por ahora, la técnica más simple es copiar y pegar las líneas que queremos ejecutar desde nuestro script hasta la consola. A partir de ahora, ya no utilizaremos editores de texto como bloc de notas, sino editores especializados para la creación de scripts R. Será el objetivo del siguiente capítulo.

## 5.5 Conclusión

Felicitaciones, hemos llegado al final de este primer capítulo sobre la base de R. Sabemos:

- Instalar R
- Usar R como una calculadora
- Crear **objetos** y utilizarlos para los calculos aritméticos, comparativos y logicos
- Elejir nombres pertinentes para los objetos
- Crear un nuevo **script**
- Elejir un nombre pertinente para el archivo del script
- Ejecutar el codigo de un script
- Documentar los scripts con **comentarios**
- Usar un estilo de código para que sea agradable de leer y facil de entender



## Chapter 6

# Elegir un entorno de desarrollo

### 6.1 Editores de texto y entorno de desarrollo

Hay muchos editores de texto, el capítulo anterior permitió introducir algunos de los más simples como el bloc de notas de Windows. Sin embargo, los límites de estos editores han hecho tediosa la tarea de escribir un script. De hecho, incluso estructurando su script con comentarios, sigue siendo difícil entenderlo. Aquí es donde entran los editores de texto especializados para facilitar la escritura y la lectura de scripts. El editor de texto para R más común es Rstudio, pero hay muchos más. Hacer una lista exhaustiva de todas las soluciones disponibles está más allá del alcance de este libro, por lo que nos centraremos en las tres soluciones que utilizo a diario: **Notepad++**, **Rstudio** y **Geany**. No necesita instalar más de un editor de texto. Aquí recomendamos RStudio para principiantes a R.

### 6.2 RStudio

#### 6.2.1 Instalar RStudio

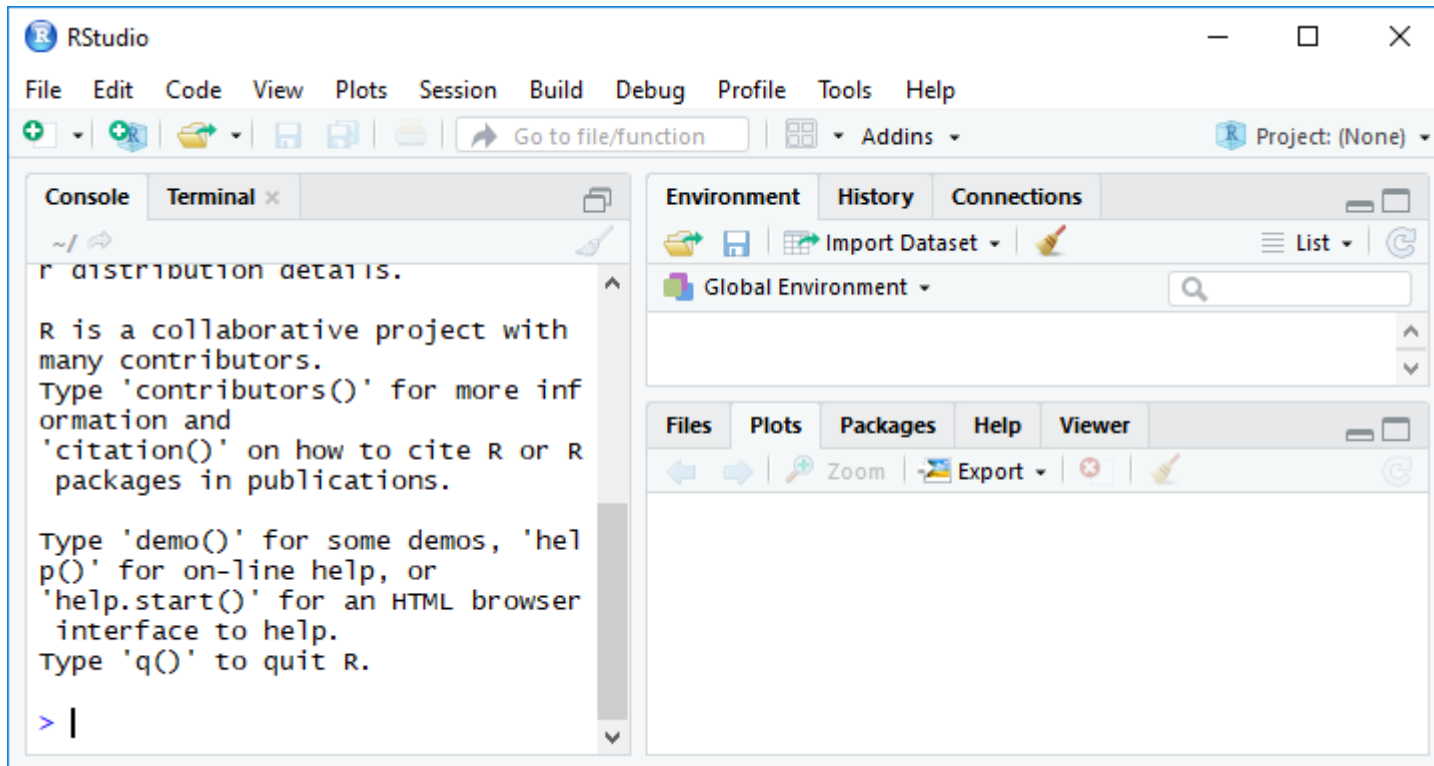
El programa para instalar el software RStudio se encuentra en la parte *Products* del sitio web de RStudio (<https://www.rstudio.com/>). Instalaremos RStudio para uso local (en nuestra computadora), por lo que la versión que nos interesa es *Desktop*. Usaremos la versión *Open Source* que es gratuita. Luego, seleccionamos la versión que corresponde a nuestro sistema operativo (Windows, Mac OS, Linux), descargamos el archivo correspondiente y lo ejecutamos para comenzar la instalación. Podemos mantener las opciones predeterminadas durante la instalación.

#### 6.2.2 Un script con RStudio

Podemos abrir RStudio. En la primera apertura, la interfaz se divide en dos con la consola R a la izquierda que vimos en el capítulo anterior (Figura 6.2). Para abrir un nuevo script, vamos al menú *Archivo* (o *File*), *Nuevo archivo* (o *New File*), *R script*. Por defecto, este archivo tiene el nombre *Untitled1*. Hemos visto en el capítulo anterior la importancia de dar un nombre pertinente a nuestros scripts, por lo que lo cambiaremos de nombre a *selecEnvDev.R*, en el menú *Archivo* (o *File*), con la opción *Guardar*



**Figure 6.1:** Logo RStudio.



**Figure 6.2:** Captura de pantalla de RStudio en Windows: pantalla por defecto.

como ... (o *Save As...*). Podemos notar que el lado izquierdo de RStudio ahora está dividido en dos, con la consola en la parte inferior de la pantalla y el script en la parte superior.

Luego podemos comenzar a escribir nuestro script con los comentarios que describan lo que vamos a encontrar allí, y agregar un cálculo simple. Una vez que hayamos copiado o escrito un código, podemos guardar nuestro script con el comando **CTRL + S** o yendo a *Archivo* (o *File*, luego *Guardar* (o *Save*)).

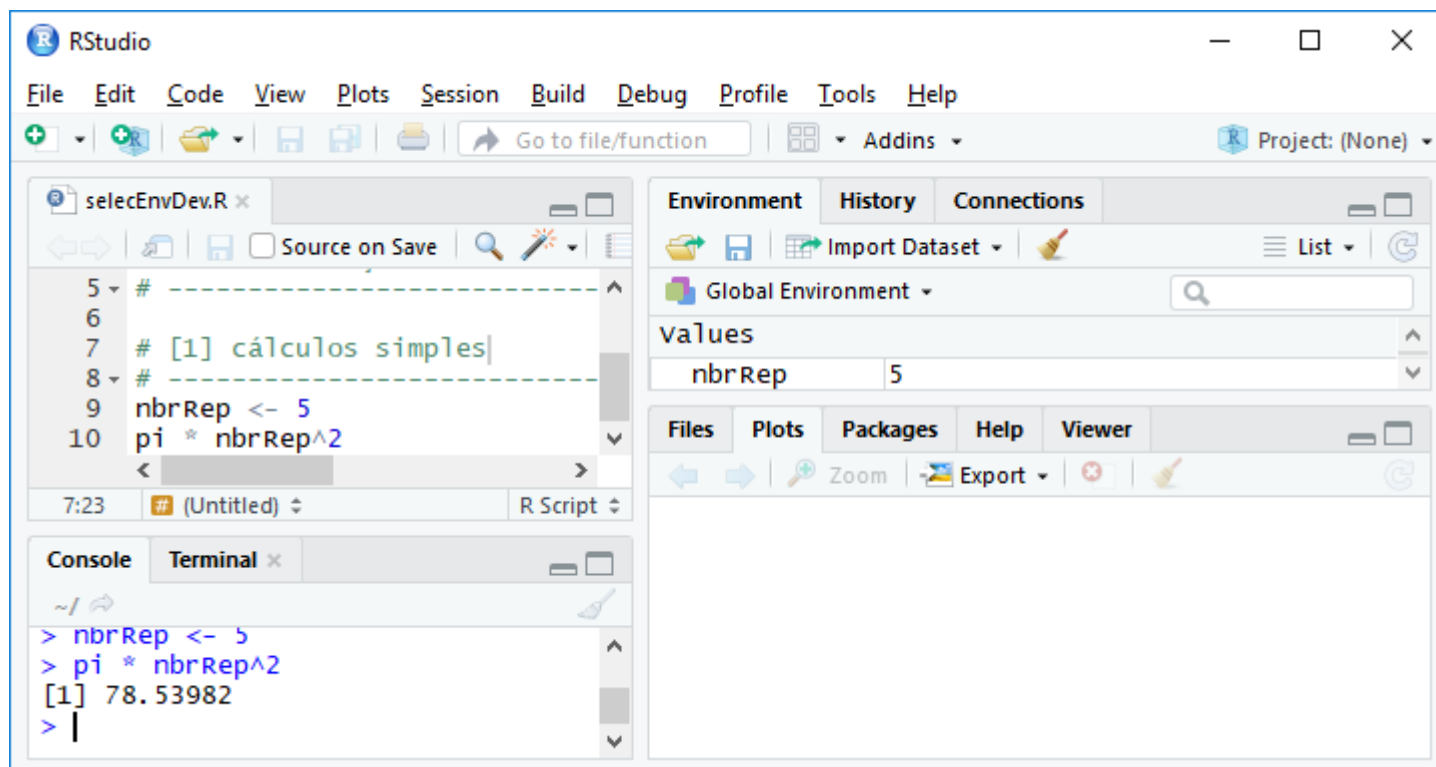
```
# -----
# Un script para seleccionar su entorno de desarrollo
# fecha de creación : 27/06/2018
# autor : François Rebaudo
# -----

# [1] cálculos simples
# -----
nbrRep <- 5
pi * nbrRep^2

## [1] 78.53982
```

Para ejecutar nuestro script, simplemente seleccionamos las líneas que deseamos ejecutar y usamos la combinación de teclas **CTRL + ENTER**. El resultado aparece en la consola (Figura 6.3).

Podemos ver que, de forma predeterminada, en la parte del script, los comentarios aparecen en verde, los números en azul y el resto del código en negro. En la parte de la consola, lo que se ejecutó aparece en azul y los resultados de la ejecución en negro. También podemos observar que en la parte del código cada línea tiene un número correspondiente al número de línea a la izquierda sobre un fondo gris. Este es el resultado de preferencias de sintaxis predeterminado con RStudio. Estas preferencias de sintaxis pueden modificarse yendo al menú *Herramientas* (o *Tools*), *Opciones globales ...* (o *Global Options...*),



**Figure 6.3:** Captura de pantalla de RStudio en Windows: ejecutar nuestro script con CTRL + ENTER.

*Aspecto* (o *Appearance*), y luego seleccionando otro tema del *Editor de tema*: (o *Editor theme*:). Elegiremos el tema *Cobalt*, luego OK (Figura 6.4).

Sabemos cómo crear un nuevo script, guardarlo, ejecutar su contenido y cambiar la apariencia de RStudio. Veremos los muchos otros beneficios de RStudio a lo largo de este libro, ya que es el entorno de desarrollo que se utilizará. Sin embargo, seremos especialmente cuidadosos de que todos los scripts desarrollados a lo largo de este libro se ejecuten de la misma manera, independientemente del entorno de desarrollo utilizado.

## 6.3 Notepad++ avec Npp2R

### 6.3.1 Instalar Notepad++ (solamente para Windows)

El programa para instalar Notepad ++ se puede encontrar en la pestaña *Downloads* (<https://notepad-plus-plus.org/download/>). Podemos elegir entre la versión de 32-bit y la de 64-bit (64-bit si no sabe qué versión elegir). Notepad++ es suficiente para escribir un script, pero es aún más poderoso con *Notepad++ to R (Npp2R)* que permite ejecutar automáticamente nuestros scripts en una consola localmente en nuestra computadora o remotamente en un servidor.

### 6.3.2 Instalar Npp2R

El programa para instalar Npp2R está alojado en el sitio de Sourceforge (<https://sourceforge.net/projects/npp2r/>). Npp2R debe instalarse después de Notepad++.



Figure 6.4: Captura de pantalla de RStudio en Windows: cambiar preferencias de sintaxis.



Figure 6.5: Logo Notepad++

### 6.3.3 Un script con Notepad++

Al abrir por primera vez, Notepad++ muestra un archivo vacío *new 1* (Figura 6.6).

Como ya hemos creado un script para probarlo con RStudio, lo abriremos de nuevo con Notepad++. En *Archivo*, seleccionamos *Abrir ...* luego elijemos el script *selecEnvDev.R* creado previamente. Una vez que el script está abierto, vamos a *Idioma*, luego *R*, y de nuevo *R*. Aparece el resaltado de sintaxis (Figura 6.7).

La ejecución del script solo se puede realizar si se está ejecutando Npp2R. Para hacerlo, es necesario ejecutar el programa Npp2R desde el prompt de Windows. Un icono debe aparecer en la parte inferior de su pantalla demostrando que Npp2R está prendido. La ejecución automática del código de Notepad++ se realiza seleccionando el código para ejecutar y luego usando el comando F8. Si el comando no funciona, puede ser necesario reiniciar la computadora. Si el comando funciona, se abrirá una nueva ventana con una consola que ejecuta las líneas deseadas (Figura 6.8).

Al igual que con RStudio, el resaltado de sintaxis se puede cambiar desde el menú *Configuración*, y se puede seleccionar un nuevo tema (por ejemplo, *Solarized* en la Figura 6.9)



Figure 6.6: Captura de pantalla de Notepad++ en Windows: pantalla por defecto.

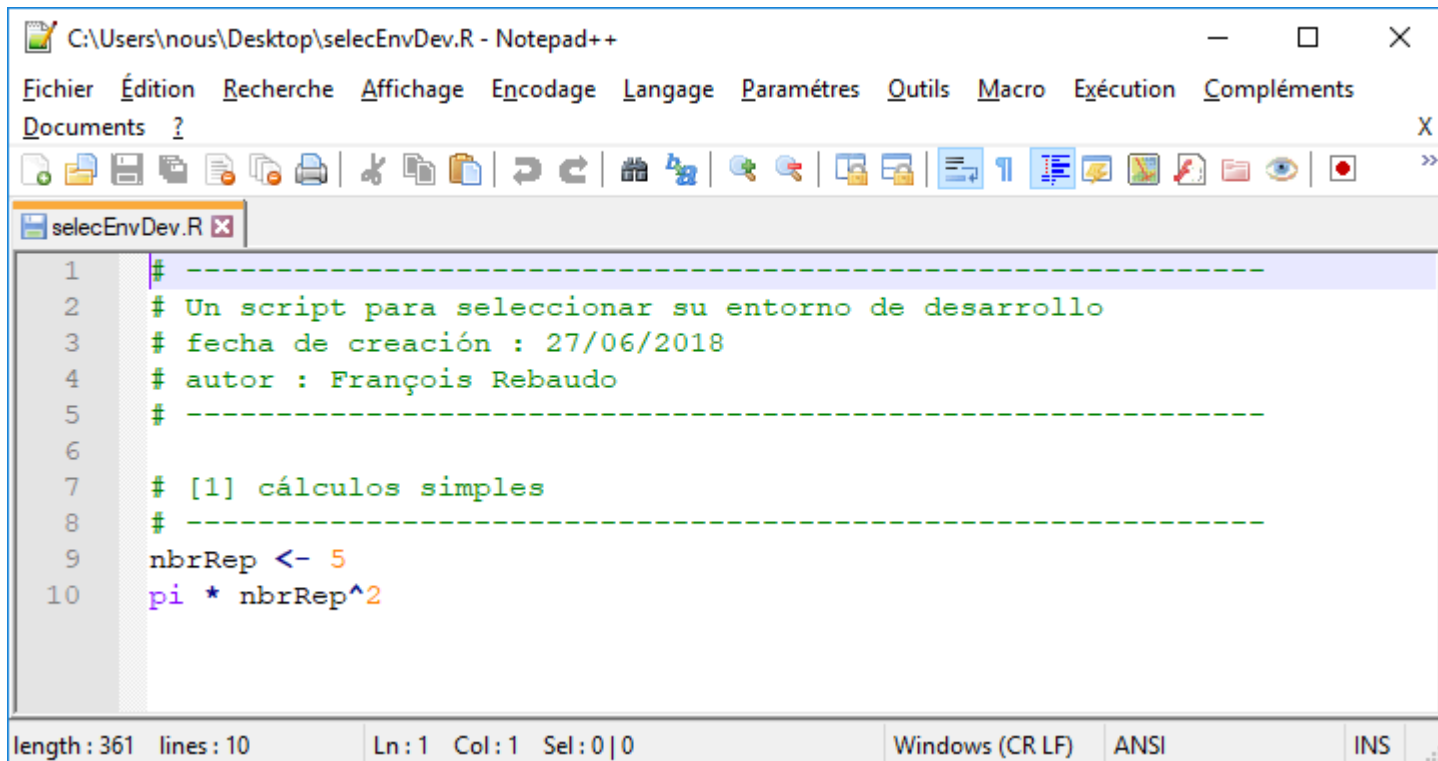


Figure 6.7: Captura de pantalla de Notepad++ en Windows: ejecutar nuestro script con F8.

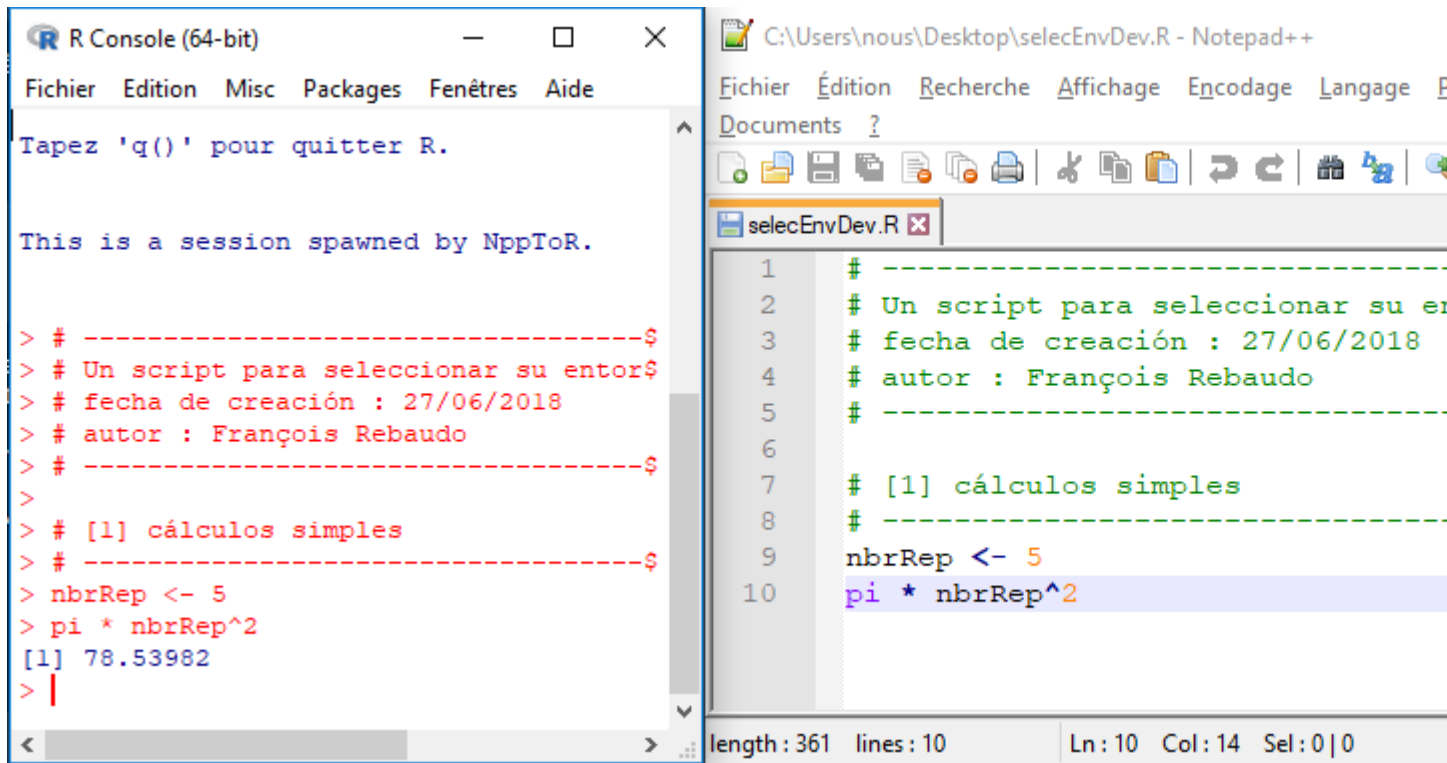


Figure 6.8: Captura de pantalla de Notepad++ en Windows: la consola con F8.



Figure 6.9: Captura de pantalla de Notepad++ en Windows con el tema Solarized.





Figure 6.10: Logo Geany

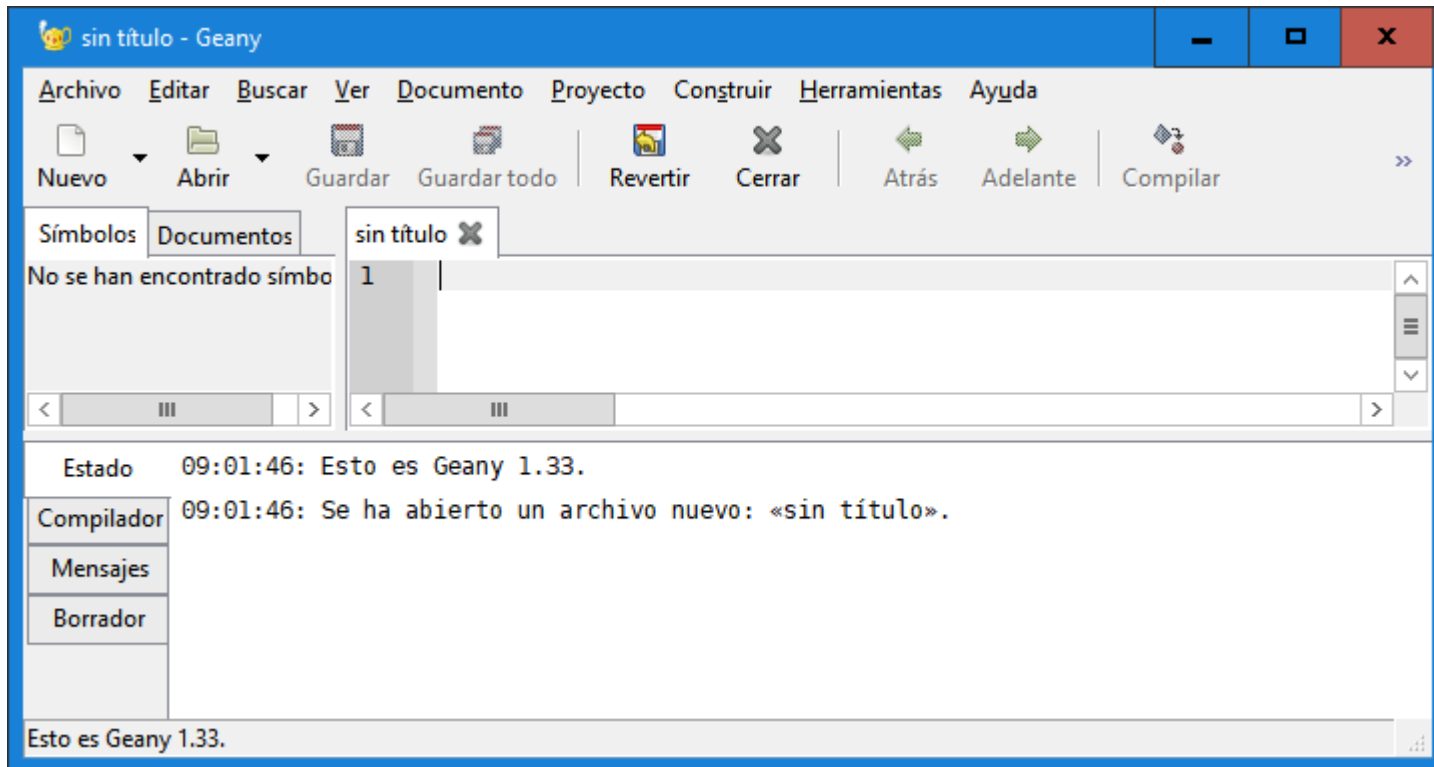


Figure 6.11: Captura de pantalla de Geany en Windows: pantalla por defecto.

Comparado con otros editores de texto, Notepad++ tiene la ventaja de ser muy liviano, rapido y ofrece una amplia gama de opciones para personalizar la escritura de códigos.

## 6.4 Geany

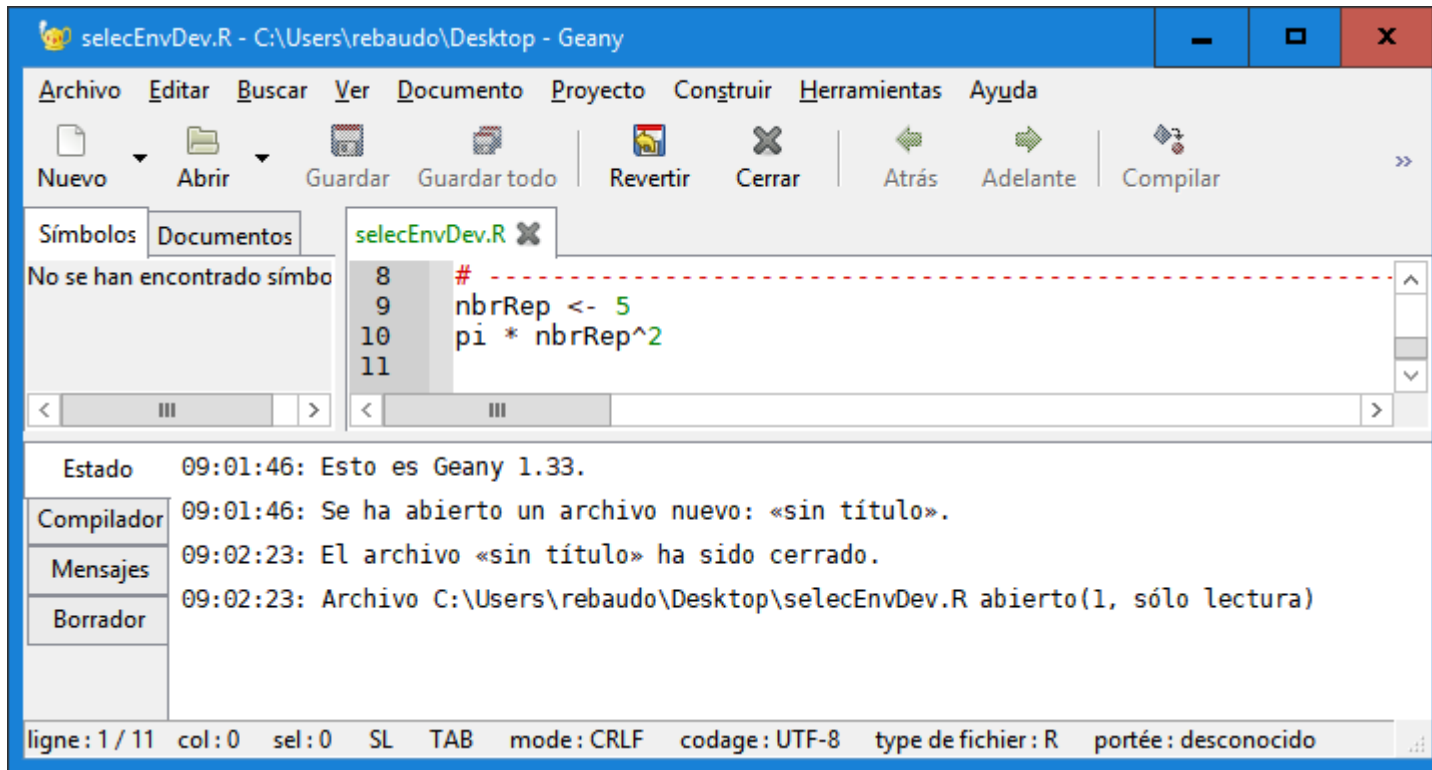
### 6.4.1 Instalar Geany (para Linux, Mac OSX y Windows)

El programa para instalar Geany se puede encontrar en la pestaña *Downloads* en el menú de la izquierda *Releases* de la página web (<https://www.geany.org/>). Luego solo descargamos el ejecutable para Windows o el dmg para Mac OSX. Los usuarios de Linux preferirán un `sudo apt-get install geany`.

### 6.4.2 Un script con Geany

Al abrir por primera vez, como para RStudio y Notepad++, se crea un archivo vacío (Figura 6.11).

Podemos abrir nuestro script con *Archivo, Abrir* (Figura 6.12).



**Figure 6.12:** Captura de pantalla de Geany en Windows: abrir un script.

Para ejecutar nuestro script, la versión de Geany para Windows no tiene un terminal incorporado, lo que hace que su uso sea limitado bajo este sistema operativo. La ejecución de un script se puede hacer abriendo R en una ventana separada y copiando y pegando las líneas que se ejecutarán. En Linux y Mac OSX, podemos abrir R en el terminal en la parte inferior de la ventana de Geany con el comando `R`. Podemos configurar Geany para una combinación de teclas para ejecutar el código seleccionado (por ejemplo `CTRL + R`). Para esto hay que permitir el envío de selección al terminal (`send_selection_unsafe = true`) in archivo `geany.conf` y elegir el comando para enviar al terminal (en *Editar, Preferencias, Combinaciones*). Para cambiar el tema de Geany, hay una colección de temas disponibles en GitHub (<https://github.com/geany/geany-themes/>). El tema se puede cambiar a través del menú *Ver, cambiar Esquema del color ...* (un ejemplo con el tema *Solarized* Figura @ref(Fig: screenCapGeany03)).

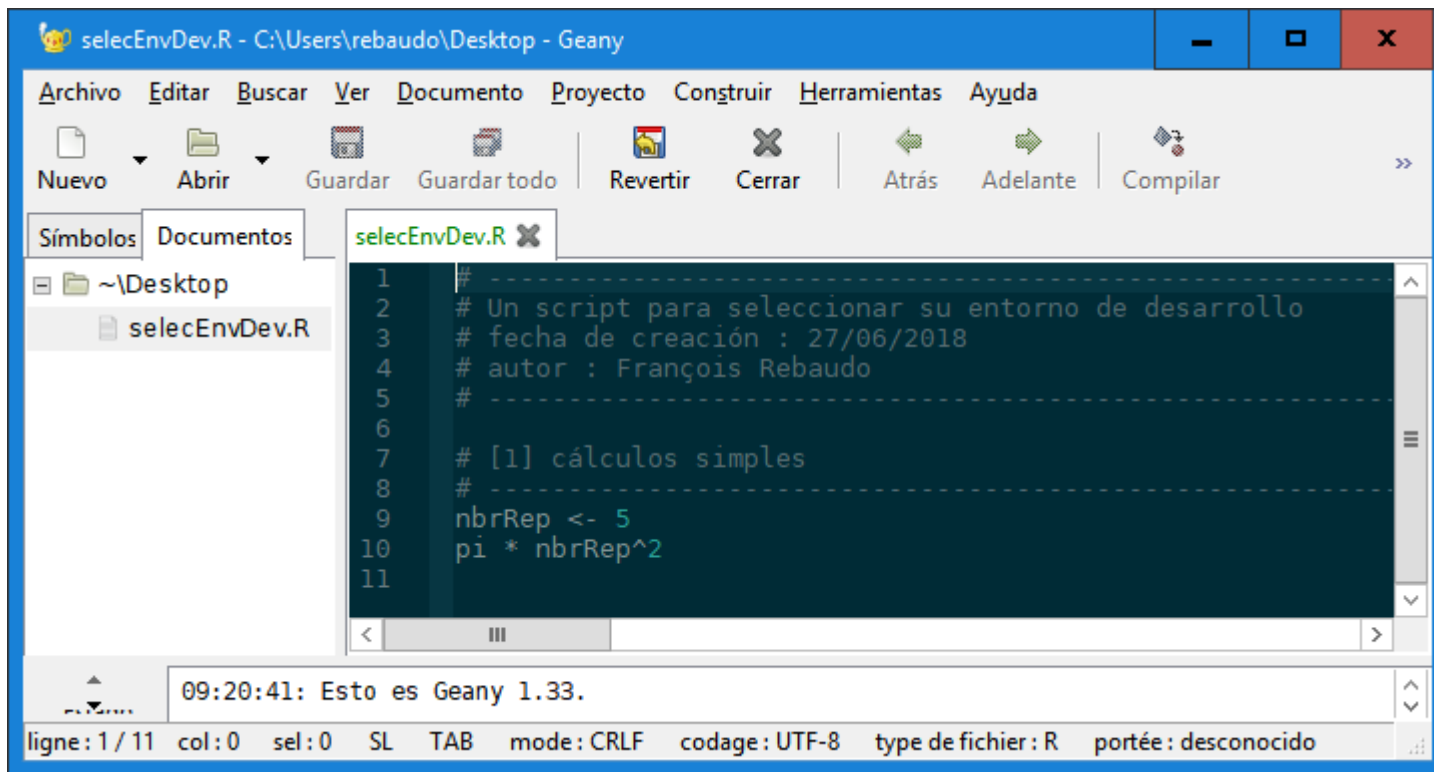
## 6.5 Otras soluciones

Hay muchas otras soluciones, algunas especializadas para R como **Tinn-R** (<https://sourceforge.net/projects/tinn-r/>), y otras más generales para programación como **Atom** (<https://atom.io/>), **Sublime Text** (<https://www.sublimetext.com/>), **Vim** (<https://www.vim.org/>), **Gedit** (<https://wiki.gnome.org/Apps/Gedit>), **GNU Emacs** (<https://www.gnu.org/software/emacs/>), **Jupyter** (<http://jupyter.org>), o **Brackets** (<http://brackets.io/>) y **Eclipse** (<http://www.eclipse.org/>).

## 6.6 Conclusión

Felicitaciones, llegamos al final de este capítulo sobre el entorno de desarrollo para el uso de R. Hasta ahora sabemos:

- Instalar RStudio, Geany o Notepad++
- Reconocer y elegir nuestro entorno de preferencia



**Figure 6.13:** Captura de pantalla de Geany en Windows: cambiar esquema de color.

A partir de acá podremos concentrarnos en el lenguaje de programación R en un ambiente, facilitando el trabajo de lectura y de escritura del código. Esto ya es un gran paso para manejar R.



## Chapter 7

# Tipos de datos

Vimos anteriormente cómo crear un objeto. Un objeto es como una caja en la que almacenaremos *información*. Hasta ahora solo hemos almacenado números, pero en este capítulo veremos que es posible almacenar otra información y nos detendremos en los tipos más comunes. En este capítulo utilizaremos **funciones** sobre las cuales volveremos más adelante.

### 7.1 El tipo `numeric`

El tipo `numeric` es lo que hemos hecho hasta ahora, almacenando números. Hay dos tipos principales de números en R: enteros (*integer*) y números decimales (*double*). Por defecto, R considera todos los números como números decimales y asigna el tipo `double`. Para verificar el tipo de datos utilizaremos la función `typeof()` que toma como *argumento* un objeto (o directamente la información que queremos probar). También podemos usar la función `is.double()` que devolverá `TRUE` si el número está en formato `double` y `FALSE` en caso contrario. La función genérica `is.numeric()` devolverá `TRUE` si el objeto está en formato `numeric` y `FALSE` en caso contrario.

```
nbrRep <- 5
typeof(nbrRep)
```

```
## [1] "double"
```

```
typeof(5.32)
```

```
## [1] "double"
```

```
is.numeric(5)
```

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

```
is.double(5)
```

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

Si queremos decirle a R que vamos a trabajar con un entero, entonces necesitamos convertir nuestro número decimal en un entero con la función `as.integer()`. También podemos usar la función `is.integer()` que devolverá `TRUE` si el número está en formato `integer` y `FALSE` en caso contrario.

```
nbrRep <- as.integer(5)
typeof(nbrRep)
```

```
## [1] "integer"
```

```
typeof(5.32)
```

```
## [1] "double"
```

```
typeof(as.integer(5.32))
```

```
## [1] "integer"
```

```
as.integer(5.32)
```

```
## [1] 5
```

```
as.integer(5.99)
```

```
## [1] 5
```

```
is.numeric(nbrRep)
```

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

Vemos aquí que convertir un número como 5.99 a `integer` solo devolverá la parte entera, 5.

```
is.integer(5)
```

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

```
is.numeric(5)
```

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

```
is.integer(as.integer(5))
```

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

```
is.numeric(as.integer(5))
```

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

La suma de un número entero `integer` y un número decimal `double` devuelve un número decimal.

```
sumIntDou <- as.integer(5) + 5.2
typeof(sumIntDou)
```

```
## [1] "double"
```

```
sumIntInt <- as.integer(5) + as.integer(5)
typeof(sumIntInt)
```

```
## [1] "integer"
```

Para resumir, el tipo `numeric` contiene dos subtipos, los tipos `integer` para enteros y el tipo `double` para los números decimales. Por defecto, R asigna el tipo `double` a los números.

Tenga cuidado, hay una trampa para usar la función `is.integer()`. No nos dice si el número es un número entero, pero si es de tipo `integer`. De hecho, uno puede almacenar un entero en una variable de tipo `double`.

Los números almacenados en una variable de tipo `integer` son codificados en 32 bits y, por lo tanto, pueden tomar valores entre 0 y  $2^{32}-1 = 4294967295$ . Hay otra forma de decirle a R que un número es un número entero, usando el sufijo `L`. Por ejemplo, `5L` es lo mismo que `as.integer(5)`. El origen del sufijo `L`, que se remonta a una época en que las computadoras usaban palabras de 16 bits y 32 bits, era un tipo `Largo`. ¡Ahora las computadoras usan palabras de 64 bits y 32 bits es bastante corta!

No podemos dejar esta sección sin mencionar las funciones `round()` `ceiling()` `trunc()` o `floor()` que devuelven la parte entera de un número, pero déjelo en el tipo `double`. Para obtener más información, podemos usar la ayuda de R con `?Ronda`.

```
{R} roundDou <- round (5.2) typeof (roundDou) '
```

## 7.2 El tipo character

El tipo `character` es texto. De hecho, R permite trabajar con texto. Para especificar a R que la información contenida en un objeto está en formato de texto (o generalmente para todos los textos), usamos las comillas dobles (`"`) o las comillas simples (`'`).

```
myText <- "azerty"
myText2 <- 'azerty'
myText3 <- 'azerty uiop qsd fg h j k l m'
typeof(myText3)
```

```
## [1] "character"
```

Tanto las comillas dobles y simples son útiles en nuestro texto. También podemos *escapar* un carácter especial como comillas gracias al signo de barra invertida `\`.

```
myText <- "a 'ze' 'rt' y"
print(myText)
```

```
## [1] "a 'ze' 'rt' y"
```

```
myText2 <- 'a "zert" y'
print(myText2)
```

```
## [1] "a \"zert\" y"
```

```
myText3 <- 'azerty uiop qsd fg h j k l m'
print(myText3)
```

```
## [1] "azerty uiop qsd fg h j k l m"
```

```
myText4 <- "qwerty \" azerty "
print(myText4)
```

```
## [1] "qwerty \" azerty "
```

```
myText5 <- "qwerty \\ azerty "
print(myText5)
```

```
## [1] "qwerty \\ azerty "
```

De forma predeterminada, cuando creamos un objeto, su contenido no es devuelto por la consola. En Internet o en muchos libros podemos encontrar el nombre del objeto en una línea para devolver sus contenidos:

```
myText <- "a 'ze' 'rt' y"
myText
```

```
## [1] "a 'ze' 'rt' y"
```

En este libro, no lo usaremos de esta manera y preferiremos el uso de la función `print()`, que permite mostrar en la consola el contenido de un objeto. El resultado es el mismo, pero el código es más fácil de leer y más explícito sobre lo que hace.

```
myText <- "a 'ze' 'rt' y"
print(myText)
```

```
## [1] "a 'ze' 'rt' y"
```

```
nbrRep <- 5
print(nbrRep)
```

```
## [1] 5
```

También podemos poner números en formato de texto, pero no debemos olvidar poner comillas para especificar el tipo `character` o usar la función `as.character()`. Una operación entre un texto y un número devuelve un error. Por ejemplo, si agregamos 10 a 5, R nos dice que un **argumento** de la **función** + no es un tipo `numeric` y que, por lo tanto, la operación no es posible. Tampoco podemos agregar texto a texto, pero veremos más adelante cómo *concatenar* dos cadenas de texto.

```
myText <- "qwerty"
typeof(myText)
```

```
## [1] "character"
```

```
myText2 <- 5
typeof(myText2)
```

```
## [1] "double"
```

```
myText3 <- "5"
typeof(myText3)
```

```
## [1] "character"
```



```
myText2 + 10
```

```
## [1] 15
```

```
as.character(5)
```

```
## [1] "5"
```

```
# myText3 + 10 # Error in myText3 + 10 : non-numeric argument to binary operator
# "a" + "b" # Error in "a" + "b" : non-numeric argument to binary operator
```

Para resumir, el tipo `character` permite el ingreso de texto, podemos reconocerlo con comillas simples o dobles.

## 7.3 El tipo factor

El tipo `factor` corresponde a los factores. Los factores son una elección dentro de una lista finita de posibilidades. Por ejemplo, los países son factores porque existe una lista finita de países en el mundo en un momento dado. Un factor puede definirse con la función `factor()` o transformarse utilizando la función `as.factor()`. Al igual que con otros tipos de datos, podemos usar la función `is.factor()` para verificar el tipo de datos. Para obtener una lista de todas las posibilidades, existe la función `levels()` (esta función tendrá más sentido cuando nos acerquemos a los tipos de contenedores de información).

```
factor01 <- factor("aaa")
print(factor01)
```

```
## [1] aaa
## Levels: aaa
```

```
typeof(factor01)
```

```
## [1] "integer"
```

```
is.factor(factor01)
```

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

```
levels(factor01)
```

```
## [1] "aaa"
```

Un factor se puede transformar en texto con la función `as.character()` pero también en número con `as.numeric()`. Al cambiar al tipo `numeric`, cada factor toma el valor de su posición en la lista de posibilidades. En nuestro caso, solo hay una posibilidad, por lo que la función `as.numeric()` devolverá 1:

```
factor01 <- factor("aaa")
as.character(factor01)
```

```
## [1] "aaa"
```

```
as.numeric(factor01)
```

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

## 7.4 El tipo logical

El tipo `logical` corresponde a los valores `TRUE` y `FALSE` (y `NA`) que ya hemos visto con los operadores de comparación.

```
aLogic <- TRUE
print(aLogic)
```

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

```
typeof(aLogic)
```

```
## [1] "logical"
```

```
is.logical(aLogic)
```

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

```
aLogic + 1
```

```
## [1] 2
```

```
as.numeric(aLogic)
```

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

```
as.character(aLogic)
```

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

## 7.5 Acerca de NA

El valor `NA` se puede usar para especificar que no hay datos o datos faltantes. Por defecto, `NA` es `logical`, pero se puede usar para texto o números.

```
print(NA)
```

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

```
typeof(NA)
```

```
## [1] "logical"
```

```
typeof(as.integer(NA))
```

```
## [1] "integer"
```

```
typeof(as.character(NA))
```

```
## [1] "character"
```

```
NA == TRUE
```

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

```
NA == FALSE
```

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

```
NA > 1
```

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

```
NA + 1
```

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

## 7.6 Conclusión

Felicitaciones, hemos llegado al final de este capítulo sobre los tipos de datos. Ahora sabemos:

- Reconocer y hacer objetos en los principales tipos de datos
- Transformar tipos de datos de un tipo a otro

Este capítulo es la base para el próximo capítulo sobre contenedores de datos.



## Chapter 8

# Contenedores de datos

Hasta ahora hemos creado objetos simples que contienen solo un valor. Sin embargo, pudimos ver que un objeto tenía atributos diferentes, como su valor, pero también el tipo de datos contenidos (e.g., `numeric`, `character`). Ahora vamos a ver que hay diferentes tipos de contenedores para almacenar datos múltiples.

### 8.1 El contenedor vector

En R, un `vector` es una combinación de datos con la particularidad de que todos los datos contenidos en un `vector` son del mismo tipo. Podemos almacenar por ejemplo múltiples elementos del tipo `character` o `numeric` en un `vector`, pero no ambos. El contenedor `vector` es importante porque es el elemento básico de R.

#### 8.1.1 Crear un vector

Para crear un `vector` utilizaremos la función `c()` que permite combinar elementos en un `vector`. Los elementos para combinar deben estar separados por comas.

```
miVec01 <- c(1, 2, 3, 4) # un vector de 4 elementos de tipo numeric ; double
print(miVec01)
```

```
## [1] 1 2 3 4
```

```
typeof(miVec01)
```

```
## [1] "double"
```

```
is.vector(miVec01)
```

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

La función `is.vector()` permite verificar el tipo de contenedor.

```
miVec02 <- c("a", "b", "c")
print(miVec02)
```

```
## [1] "a" "b" "c"
```

```
typeof(miVec02)
```

```
## [1] "character"
```

```
is.vector(miVec02)
```

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

```
miVec03 <- c(TRUE, FALSE, FALSE, TRUE)  
print(miVec03)
```

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

```
typeof(miVec03)
```

```
## [1] "logical"
```

```
is.vector(miVec03)
```

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

```
miVecNA <- c(1, NA, 3, NA, 5)  
print(miVecNA)
```

```
## [1] 1 NA 3 NA 5
```

```
typeof(miVecNA)
```

```
## [1] "double"
```

```
is.vector(miVecNA)
```

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

```
miVec04 <- c(1, "a")  
print(miVec04)
```

```
## [1] "1" "a"
```

```
typeof(miVec04)
```

```
## [1] "character"
```

```
is.vector(miVec04)
```

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

Si combinamos diferentes tipos de datos, R intentará transformar los elementos en un tipo de forma predeterminada. Si como aquí en el objeto `miVec03` tenemos los tipos `character` y `numeric`, R convertirá todos los elementos en `character`.

```
miVec05 <- c(factor("abc"), "def")
print(miVec05)
```

```
## [1] "1" "def"
```

```
typeof(miVec05)
```

```
## [1] "character"
```

```
miVec06 <- c(TRUE, "def")
print(miVec06)
```

```
## [1] "TRUE" "def"
```

```
typeof(miVec06)
```

```
## [1] "character"
```

```
miVec07 <- c(factor("abc"), 55)
print(miVec07)
```

```
## [1] 1 55
```

```
typeof(miVec07)
```

```
## [1] "double"
```

```
miVec08 <- c(TRUE, 55)
print(miVec08)
```

```
## [1] 1 55
```

```
typeof(miVec08)
```

```
## [1] "double"
```

También podemos combinar objetos existentes dentro de un vector.

```
miVec09 <- c(miVec02, "d", "e", "f")
print(miVec09)
```

```
## [1] "a" "b" "c" "d" "e" "f"
```

```
miVec10 <- c("aaa", "aa", miVec09, "d", "e", "f")
print(miVec10)
```

```
## [1] "aaa" "aa" "a" "b" "c" "d" "e" "f" "d" "e" "f"
```

```
miVec11 <- c(789, miVec01, 564)
print(miVec11)
```

```
## [1] 789 1 2 3 4 564
```

### 8.1.2 Hacer operaciones con un vector

También podemos realizar operaciones en un vector.

```
print(miVec01)
```

```
## [1] 1 2 3 4
```

```
miVec01 + 1
```

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

```
miVec01 - 1
```

```
## [1] 0 1 2 3
```

```
miVec01 * 2
```

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

```
miVec01 /10
```

```
## [1] 0.1 0.2 0.3 0.4
```

Las operaciones de un vector a otro también son posibles, pero se debe tener cuidado para asegurar que el número de elementos en un vector sea el mismo que el otro, de lo contrario R realizará el cálculo comenzando desde el inicio del vector mas pequeño. Aquí hay un ejemplo para ilustrar lo que R hace:

```
miVec12 <- c(1, 1, 1, 1, 1, 1, 1, 1, 1)
print(miVec12)
```

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

```
miVec13 <- c(10, 20, 30)
print(miVec13)
```

```
## [1] 10 20 30
```

```
miVec12 + miVec13 # vectores de diferentes tamaños: atención al resultado
```

```
## [1] 11 21 31 11 21 31 11 21 31
```

```
miVec14 <- c(10, 20, 30, 40, 50, 60, 70, 80, 90)
print(miVec14)
```

```
## [1] 10 20 30 40 50 60 70 80 90
```

```
miVec12 + miVec14 # los vectores tienen el mismo tamaño
```



```
## [1] 11 21 31 41 51 61 71 81 91
```

```
miVec15 <- c(1, 1, 1, 1)
print(miVec15)
```

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

```
miVec15 + miVec13 # vectores de diferentes tamaños y no múltiples
```

```
## Warning in miVec15 + miVec13: la taille d'un objet plus long n'est pas
## multiple de la taille d'un objet plus court
```

```
## [1] 11 21 31 11
```

### 8.1.3 Acceder a los valores de un vector

Suele pasar que sea necesario poder acceder a los valores de un vector, es decir, recuperar un valor o un grupo de valores dentro de un vector. Para acceder a un elemento de un vector usamos los corchetes []. Entre los corchetes, podemos usar un número correspondiente al número del elemento en el vector.

```
miVec20 <- c(10, 20, 30, 40, 50, 60, 70, 80, 90)
miVec21 <- c("a", "b", "c", "d", "e", "f", "g", "h", "i")
print(miVec20)
```

```
## [1] 10 20 30 40 50 60 70 80 90
```

```
print(miVec21)
```

```
## [1] "a" "b" "c" "d" "e" "f" "g" "h" "i"
```

```
print(miVec20[1])
```

```
## [1] 10
```

```
print(miVec21[3])
```

```
## [1] "c"
```

También podemos usar la combinación de diferentes elementos (otro vector).

```
print(miVec20[c(1, 5, 9)])
```

```
## [1] 10 50 90
```

```
print(miVec21[c(4, 3, 1)])
```

```
## [1] "d" "c" "a"
```

```
print(miVec21[c(4, 4, 3, 4, 3, 2, 5)])
```

```
## [1] "d" "d" "c" "d" "c" "b" "e"
```

También podemos seleccionar elementos usando un operador de comparación o un operador lógico.

```
print(miVec20[miVec20 >= 50])
```

```
## [1] 50 60 70 80 90
```

```
print(miVec20[(miVec20 >= 50) & ((miVec20 < 80))])
```

```
## [1] 50 60 70
```

```
print(miVec20[miVec20 != 50])
```

```
## [1] 10 20 30 40 60 70 80 90
```

```
print(miVec20[miVec20 == 30])
```

```
## [1] 30
```

```
print(miVec20[(miVec20 == 30) | (miVec20 == 50)])
```

```
## [1] 30 50
```

```
print(miVec21[miVec21 == "a"])
```

```
## [1] "a"
```

Otra característica interesante es la posibilidad de condicionar los elementos a seleccionar en base a otro vector.

```
print(miVec21[miVec20 >= 50])
```

```
## [1] "e" "f" "g" "h" "i"
```

```
print(miVec21[(miVec20 >= 50) & ((miVec20 < 80))])
```

```
## [1] "e" "f" "g"
```

```
print(miVec21[miVec20 != 50])
```

```
## [1] "a" "b" "c" "d" "f" "g" "h" "i"
```

```
print(miVec21[miVec20 == 30])
```

```
## [1] "c"
```

```
print(miVec21[(miVec20 == 30) | (miVec20 == 50)])
```

```
## [1] "c" "e"
```

```
print(miVec21[(miVec20 == 30) | (miVec21 == "h")])
```

```
## [1] "c" "h"
```

También es posible excluir ciertos elementos en lugar de seleccionarlos.

```
print(miVec20[-1])
```

```
## [1] 20 30 40 50 60 70 80 90
```

```
print(miVec21[-5])
```

```
## [1] "a" "b" "c" "d" "f" "g" "h" "i"
```

```
print(miVec20[-c(1, 2, 5)])
```

```
## [1] 30 40 60 70 80 90
```

```
print(miVec21[-c(1, 2, 5)])
```

```
## [1] "c" "d" "f" "g" "h" "i"
```

Los elementos de un vector también se pueden seleccionar sobre la base de un vector tipo logical. En este caso, solo se seleccionarán elementos con un valor TRUE.

```
miVec22 <- c(TRUE, TRUE, FALSE, TRUE, FALSE, TRUE, FALSE, TRUE, TRUE)
print(miVec21[miVec22])
```

```
## [1] "a" "b" "d" "f" "h" "i"
```

### 8.1.4 Dar nombres a los elementos de un vector

Los elementos de un vector se pueden nombrar para referenciarlos y luego seleccionarlos. La función `names()` recupera los nombres de los elementos de un vector.

```
miVec23 <- c(aaa = 10, bbb = 20, ccc = 30, ddd = 40, eee = 50)
print(miVec23)
```

```
## aaa bbb ccc ddd eee
## 10 20 30 40 50
```

```
print(miVec23["bbb"])
```

```
## bbb
## 20
```

```
print(miVec23[c("bbb", "ccc", "bbb")])
```

```
## bbb ccc bbb
## 20 30 20
```

```
names(miVec23)
```

```
## [1] "aaa" "bbb" "ccc" "ddd" "eee"
```

### 8.1.5 Editar los elementos de un vector

Para modificar un vector, operamos de la misma manera que para modificar un objeto simple, con el signo `<-` y el elemento o los elementos a modificar entre corchetes.

```
print(miVec21)
```

```
## [1] "a" "b" "c" "d" "e" "f" "g" "h" "i"
```

```
miVec21[3] <- "zzz"
print(miVec21)
```

```
## [1] "a" "b" "zzz" "d" "e" "f" "g" "h" "i"
```

```
miVec21[(miVec20 >= 50) & ((miVec20 < 80))] <- "qwerty"
print(miVec21)
```

```
## [1] "a" "b" "zzz" "d" "qwerty" "qwerty" "qwerty" "h"
## [9] "i"
```

```
print(miVec23)
```

```
## aaa bbb ccc ddd eee
## 10 20 30 40 50
```

```
miVec23["ccc"] <- miVec23["ccc"] + 100
print(miVec23)
```

```
## aaa bbb ccc ddd eee
## 10 20 130 40 50
```

También podemos cambiar los nombres asociados con los elementos de un vector.

```
print(miVec23)
```

```
## aaa bbb ccc ddd eee
## 10 20 130 40 50
```

```
names(miVec23)[2] <- "bb_bb"
print(miVec23)
```

```
##   aaa bb_bb   ccc   ddd   eee
##   10    20  130   40   50
```

Podemos hacer mucho más con un vector y volveremos a su manejo y operaciones posibles en el capítulo sobre funciones.

## 8.2 El contenedor list

El segundo tipo de contenedor que vamos a presentar es el contenedor `list`, que es también el segundo contenedor después del tipo vector debido a su importancia en la programación con R. El contenedor de tipo `list` le permite almacenar **listas** de elementos. Contrariamente a lo que vimos antes con el tipo vector, los elementos del tipo `list` pueden ser diferentes (por

ejemplo, un vector de tipo `numeric`, luego un vector de tipo `character`). Los elementos del tipo `list` también pueden ser contenedores diferentes (por ejemplo, un vector, luego una `list`). El tipo de contenedor `list` tendrá mas sentido cuando hayamos estudiado los **bucles** y **funciones** de la familia `apply`.

### 8.2.1 Crear una list

Para crear una `list` usaremos la función `list()`, que toma elementos (objetos) como argumentos.

```
miList01 <- list()
print(miList01)
```

```
## list()
```

```
miList02 <- list(5, "qwerty", c(4, 5, 6), c("a", "b", "c"))
print(miList02)
```

```
## [[1]]
## [1] 5
##
## [[2]]
## [1] "qwerty"
##
## [[3]]
## [1] 4 5 6
##
## [[4]]
## [1] "a" "b" "c"
```

```
miList03 <- list(5, "qwerty", list(c(4, 5, 6), c("a", "b", "c")))
print(miList03)
```

```
## [[1]]
## [1] 5
##
## [[2]]
## [1] "qwerty"
##
## [[3]]
## [[3]][[1]]
## [1] 4 5 6
##
## [[3]][[2]]
## [1] "a" "b" "c"
```

La función `is.list()` se usa para probar si hemos creado un objeto de tipo `list`.

```
is.list(miList02)
```

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

```
typeof(miList02)
```

```
## [1] "list"
```

### 8.2.2 Acceder a los valores de una list

Los elementos del contenedor `list` son identificables por los corchetes dobles `[[ ]]`.

```
print(miList02)
```

```
## [[1]]  
## [1] 5  
##  
## [[2]]  
## [1] "qwerty"  
##  
## [[3]]  
## [1] 4 5 6  
##  
## [[4]]  
## [1] "a" "b" "c"
```

En el objeto de tipo `list` `miList02`, hay cuatro elementos identificables con `[[1]]`, `[[2]]`, `[[3]]` y `[[4]]`. Cada uno de los elementos es de tipo `vector`. El primer elemento tiene un tamaño de 1 con elementos del tipo `double`, el segundo elemento tiene un tamaño de 1 con elementos del tipo `character`, el tercero elemento tiene un tamaño de 3 con elementos del tipo `double`, y el cuarto elemento tiene un tamaño de 3 con elementos del tipo `character`.

```
typeof(miList02)
```

```
## [1] "list"
```

```
print(miList02[[1]])
```

```
## [1] 5
```

```
typeof(miList02[[1]])
```

```
## [1] "double"
```

```
print(miList02[[2]])
```

```
## [1] "qwerty"
```

```
typeof(miList02[[2]])
```

```
## [1] "character"
```

```
print(miList02[[3]])
```

```
## [1] 4 5 6
```

```
typeof(miList02[[3]])
```

```
## [1] "double"
```

```
print(miList02[[4]])
```

```
## [1] "a" "b" "c"
```

```
typeof(miList02[[4]])
```

```
## [1] "character"
```

El acceso al segundo elemento del vector ubicado en la cuarta posición de la list se hace con `miList02[[4]][2]`. Usamos doble corchete para el cuarto elemento de la list, luego corchetes simples para el segundo elemento del vector.

```
print(miList02[[4]][2])
```

```
## [1] "b"
```

Como una list puede contener una o más list, podemos acceder a la información buscada combinando corchetes dobles. El objeto `miList04` es una list de dos elementos: la list `miList02` y la list `miList03`. El objeto `miList03` en sí contiene una list como tercer elemento. Para acceder al primer elemento del vector en la primera posición del elemento en la tercera posición del segundo elemento del list `miList04`, podemos usar `miList04[[2]][[3]][[1]][1]`. No hay límite en cuanto a la profundidad de list pero en la práctica raramente hay necesidad de hacer list de list de list.

```
miList04 <- list(miList02, miList03)
print(miList04)
```

```
## [[1]]
## [[1]][[1]]
## [1] 5
##
## [[1]][[2]]
## [1] "qwerty"
##
## [[1]][[3]]
## [1] 4 5 6
##
## [[1]][[4]]
## [1] "a" "b" "c"
##
##
## [[2]]
## [[2]][[1]]
## [1] 5
##
## [[2]][[2]]
## [1] "qwerty"
##
## [[2]][[3]]
## [[2]][[3]][[1]]
## [1] 4 5 6
##
## [[2]][[3]][[2]]
## [1] "a" "b" "c"
```

```
print(miList04[[2]][[3]][[1]][1])
```

```
## [1] 4
```

Para concretar el ejemplo anterior, podemos imaginar especies de barrenadores del maíz (*Sesamia nonagrioides* y *Ostrinia nubilalis*), muestreados en diferentes sitios, con diferentes abundancias en cuatro fechas. Aquí daremos nombres a los elementos de las `list`.

```
bddInsect <- list(Snonagrioides = list(site01 = c(12, 5, 8, 7), site02 = c(5, 23, 4, 41), site03 = c(12, 0, 0, 0)),
  Onubilalis = list(site01 = c(12, 1, 2, 3), site02 = c(0, 0, 0, 1), site03 = c(1, 1, 2, 3)))
print(bddInsect)
```

```
## $Snonagrioides
## $Snonagrioides$site01
## [1] 12 5 8 7
##
## $Snonagrioides$site02
## [1] 5 23 4 41
##
## $Snonagrioides$site03
## [1] 12 0 0 0
##
##
## $Onubilalis
## $Onubilalis$site01
## [1] 12 1 2 3
##
## $Onubilalis$site02
## [1] 0 0 0 1
##
## $Onubilalis$site03
## [1] 1 1 2 3
```

Leer una larga línea de código como la línea para crear el objeto `bddInsect` resulta difícil porque la profundidad de los elementos solo se puede deducir de los paréntesis. Es por eso que vamos a reorganizar el código para que sea más legible mediante el **margen adicional**. El margen adicional implica poner información en diferentes niveles para que podamos identificar rápidamente los diferentes niveles de un código. Para aplicar el margen adicional se presiona la tecla de tabulación. Volveremos al margen adicional con más detalles en el capítulo sobre **bucles**. Recordemos por el momento que si una línea de código es demasiado larga, podemos saltar de línea y usar el margen adicional. R leerá todo como una sola línea de código.

```
bddInsect <- list(
  Snonagrioides = list(
    site01 = c(12, 5, 8, 7),
    site02 = c(5, 23, 4, 41),
    site03 = c(12, 0, 0, 0)
  ),
  Onubilalis = list(
    site01 = c(12, 1, 2, 3),
    site02 = c(0, 0, 0, 1),
    site03 = c(1, 1, 2, 3)
  )
)
```

Podemos seleccionar los datos de abundancia del segundo sitio de la primera especie como previamente `bddInsect[[1]][[2]]`, o alternativamente usando los nombres de los elementos `bddInsect$Snonagrioides$site02`. Para hacer esto usamos el



signo \$, o como alternativa el nombre de los elementos con comillas simples o dobles `bddInsect[['Snonagrioides']][['sitio02']]`.

```
print(bddInsect[[1]][[2]])
```

```
## [1] 5 23 4 41
```

```
print(bddInsect$Snonagrioides$site02)
```

```
## [1] 5 23 4 41
```

```
print(bddInsect[['Snonagrioides']][['site02']])
```

```
## [1] 5 23 4 41
```

En cuanto a los vectores, podemos recuperar los nombres de los elementos con la función `names()`.

```
names(bddInsect)
```

```
## [1] "Snonagrioides" "Onubilalis"
```

```
names(bddInsect[[1]])
```

```
## [1] "site01" "site02" "site03"
```

Cuando usamos los corchetes dobles `[[ ]]` o el signo \$, R devuelve el contenido del elemento seleccionado. En nuestro ejemplo, los datos de abundancia están contenidos como un vector, por lo que R devuelve un elemento del tipo `vector`. Si queremos seleccionar un elemento de una `list` pero manteniendo el formato `list`, entonces podemos usar corchetes simples `[ ]`.

```
print(bddInsect[[1]][[2]])
```

```
## [1] 5 23 4 41
```

```
typeof(bddInsect[[1]][[2]])
```

```
## [1] "double"
```

```
is.list(bddInsect[[1]][[2]])
```

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

```
print(bddInsect[[1]][2])
```

```
## $site02
## [1] 5 23 4 41
```

```
typeof(bddInsect[[1]][2])
```

```
## [1] "list"
```

```
is.list(bddInsect[[1]][2])
```

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

El uso de corchetes simples `[]` es útil cuando queremos recuperar varios elementos de una `list`. Por ejemplo, para seleccionar las abundancias de insectos de los primeros dos sitios de la primera especie, usaremos `bddInsect [[1]][c(1, 2)]` o alternativamente `bddInsect [[1]][c("site01", "sitio02")]`.

```
print(bddInsect[[1]][c(1, 2)])
```

```
## $site01
## [1] 12  5  8  7
##
## $site02
## [1]  5 23  4 41
```

```
print(bddInsect[[1]][c("site01", "site02")])
```

```
## $site01
## [1] 12  5  8  7
##
## $site02
## [1]  5 23  4 41
```

### 8.2.3 Editar una list

Una `list` se puede modificar de la misma manera que para el contenedor `vector`, es decir, haciendo referencia con corchetes al elemento que queremos modificar.

```
print(miList02)
```

```
## [[1]]
## [1] 5
##
## [[2]]
## [1] "qwerty"
##
## [[3]]
## [1] 4 5 6
##
## [[4]]
## [1] "a" "b" "c"
```

```
miList02[[1]] <- 12
print(miList02)
```

```
## [[1]]
## [1] 12
##
## [[2]]
## [1] "qwerty"
##
## [[3]]
## [1] 4 5 6
##
## [[4]]
```

```
## [1] "a" "b" "c"
```

```
miList02[[4]] <- c("d", "e", "f")
print(miList02)
```

```
## [[1]]
## [1] 12
##
## [[2]]
## [1] "qwerty"
##
## [[3]]
## [1] 4 5 6
##
## [[4]]
## [1] "d" "e" "f"
```

```
miList02[[4]] <- c("a", "b", "c", miList02[[4]], "g", "h", "i")
print(miList02)
```

```
## [[1]]
## [1] 12
##
## [[2]]
## [1] "qwerty"
##
## [[3]]
## [1] 4 5 6
##
## [[4]]
## [1] "a" "b" "c" "d" "e" "f" "g" "h" "i"
```

```
miList02[[4]][5] <- "eee"
print(miList02)
```

```
## [[1]]
## [1] 12
##
## [[2]]
## [1] "qwerty"
##
## [[3]]
## [1] 4 5 6
##
## [[4]]
## [1] "a" "b" "c" "d" "eee" "f" "g" "h" "i"
```

```
miList02[[3]] <- miList02[[3]] * 10 - 1
print(miList02)
```

```
## [[1]]
## [1] 12
##
```

```
## [[2]]
## [1] "qwerty"
##
## [[3]]
## [1] 39 49 59
##
## [[4]]
## [1] "a"    "b"    "c"    "d"    "eee"  "f"    "g"    "h"    "i"
```

```
miList02[[3]][2] <- miList02[[1]] * 100
print(miList02)
```

```
## [[1]]
## [1] 12
##
## [[2]]
## [1] "qwerty"
##
## [[3]]
## [1] 39 1200 59
##
## [[4]]
## [1] "a"    "b"    "c"    "d"    "eee"  "f"    "g"    "h"    "i"
```

```
print(bddInsect)
```

```
## $Snonagrioides
## $Snonagrioides$site01
## [1] 12 5 8 7
##
## $Snonagrioides$site02
## [1] 5 23 4 41
##
## $Snonagrioides$site03
## [1] 12 0 0 0
##
##
## $Onubilalis
## $Onubilalis$site01
## [1] 12 1 2 3
##
## $Onubilalis$site02
## [1] 0 0 0 1
##
## $Onubilalis$site03
## [1] 1 1 2 3
```

```
bddInsect[['Snonagrioides']][['site02']] <- c(2, 4, 6, 8)
print(bddInsect)
```

```
## $Snonagrioides
## $Snonagrioides$site01
## [1] 12 5 8 7
```

```
##
## $Snonagrioides$site02
## [1] 2 4 6 8
##
## $Snonagrioides$site03
## [1] 12 0 0 0
##
##
## $Onubilalis
## $Onubilalis$site01
## [1] 12 1 2 3
##
## $Onubilalis$site02
## [1] 0 0 0 1
##
## $Onubilalis$site03
## [1] 1 1 2 3
```

Para combinar dos list, simplemente usamos la función `c()` que hemos usado para crear un vector.

```
miList0203 <- c(miList02, miList03)
print(miList0203)
```

```
## [[1]]
## [1] 12
##
## [[2]]
## [1] "qwerty"
##
## [[3]]
## [1] 39 1200 59
##
## [[4]]
## [1] "a" "b" "c" "d" "eee" "f" "g" "h" "i"
##
## [[5]]
## [1] 5
##
## [[6]]
## [1] "qwerty"
##
## [[7]]
## [[7]][[1]]
## [1] 4 5 6
##
## [[7]][[2]]
## [1] "a" "b" "c"
```

Un objeto de tipo `list` se puede transformar en vector con la función `unlist()` si el formato de los elementos de la lista lo permite (un vector solo puede contener elementos del mismo tipo).

```
miList05 <- list("a", c("b", "c"), "d")
print(miList05)
```

```
## [[1]]
## [1] "a"
##
## [[2]]
## [1] "b" "c"
##
## [[3]]
## [1] "d"
```

```
miVec24 <- unlist(miList05)
print(miVec24)
```

```
## [1] "a" "b" "c" "d"
```

```
miList06 <- list(c(1, 2, 3), c(4, 5, 6, 7), 8, 9, c(10, 11))
print(miList06)
```

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

```
miVec25 <- unlist(miList06)
print(miVec25)
```

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

Para agregar un elemento a una `list`, podemos usar la función `c()` o los corchetes dobles `[[ ]]`.

```
print(miList05)
```

```
## [[1]]
## [1] "a"
##
## [[2]]
## [1] "b" "c"
##
## [[3]]
## [1] "d"
```

```
miList05 <- c(miList05, "e")
print(miList05)
```

```
## [[1]]
## [1] "a"
##
## [[2]]
## [1] "b" "c"
##
## [[3]]
## [1] "d"
##
## [[4]]
## [1] "e"
```

```
miList05[[5]] <- c("fgh", "ijk")
print(miList05)
```

```
## [[1]]
## [1] "a"
##
## [[2]]
## [1] "b" "c"
##
## [[3]]
## [1] "d"
##
## [[4]]
## [1] "e"
##
## [[5]]
## [1] "fgh" "ijk"
```

Para eliminar un elemento de una list, la técnica más rápida es establecer NULL en el elemento que deseamos eliminar.

```
print(miList05)
```

```
## [[1]]
## [1] "a"
##
## [[2]]
## [1] "b" "c"
##
## [[3]]
## [1] "d"
##
## [[4]]
## [1] "e"
##
## [[5]]
## [1] "fgh" "ijk"
```

```
miList05[[2]] <- NULL
print(miList05)
```

```
## [[1]]
## [1] "a"
```

```
##
## [[2]]
## [1] "d"
##
## [[3]]
## [1] "e"
##
## [[4]]
## [1] "fgh" "ijk"
```

## 8.3 El contenedor `data.frame`

El contenedor `data.frame` se puede comparar a una *tabla*. Este es en realidad un caso especial de `list` donde todos los elementos de la `list` tienen el mismo tamaño.

### 8.3.1 Crear un `data.frame`

Para crear un `data.frame` usamos la función `data.frame()` que toma como argumentos los elementos de la tabla que queremos crear. Los elementos son del tipo vector y son todos del mismo tamaño. Podemos dar un nombre a cada *columna* (vector) de nuestra *tabla* (`data.frame`).

```
# crear un data.frame
miDf01 <- data.frame(
  numbers = c(1, 2, 3, 4),
  logicals = c(TRUE, TRUE, FALSE, TRUE),
  characters = c("a", "b", "c", "d")
)
print(miDf01)
```

```
##   numbers logicals characters
## 1      1      TRUE         a
## 2      2      TRUE         b
## 3      3     FALSE         c
## 4      4      TRUE         d
```

```
# crear vectores, y el data.frame
numbers <- c(1, 2, 3, 4)
logicals <- c(TRUE, TRUE, FALSE, TRUE)
characters <- c("a", "b", "c", "d")
miDf01 <- data.frame(numbers, logicals, characters)
print(miDf01)
```

```
##   numbers logicals characters
## 1      1      TRUE         a
## 2      2      TRUE         b
## 3      3     FALSE         c
## 4      4      TRUE         d
```



### 8.3.2 Acceder a los elementos de un data.frame

El acceso a los diferentes valores de un data.frame se puede hacer de la misma manera que para un contenedor de tipo list.

```
print(miDf01$numbers) # vector
```

```
## [1] 1 2 3 4
```

```
print(miDf01[[1]]) # vector
```

```
## [1] 1 2 3 4
```

```
print(miDf01[1]) # list
```

```
##   numbers
## 1      1
## 2      2
## 3      3
## 4      4
```

```
print(miDf01["numbers"]) # list
```

```
##   numbers
## 1      1
## 2      2
## 3      3
## 4      4
```

```
print(miDf01[["numbers"]]) # vector
```

```
## [1] 1 2 3 4
```

También podemos usar otra forma que consiste en especificar la línea o las líneas seguidas de una coma (con un espacio después de la coma), y luego la columna o columnas entre corchetes. Si se omite la información de línea o columna, R mostrará todas las líneas o columnas. Nuevamente podemos usar el número correspondiente a un elemento o el nombre del elemento que queremos seleccionar.

```
myRow <- 2
myCol <- 1
print(miDf01[myRow, myCol])
```

```
## [1] 2
```

```
print(miDf01[myRow, ])
```

```
##   numbers logicals characters
## 2      2      TRUE          b
```

```
print(miDf01[, myCol])
```

```
## [1] 1 2 3 4
```

```
myCol <- "numbers"
print(miDf01[, myCol])
```

```
## [1] 1 2 3 4
```

Es posible seleccionar múltiples líneas o columnas.

```
print(miDf01[, c(1, 2)])
```

```
##  numbers logicals
## 1      1      TRUE
## 2      2      TRUE
## 3      3     FALSE
## 4      4      TRUE
```

```
print(miDf01[c(2, 1), ])
```

```
##  numbers logicals characters
## 2      2      TRUE          b
## 1      1      TRUE          a
```

Como cada columna está en formato vector, también podemos hacer una selección que depende del contenido con operadores de comparación y operadores lógicos.

```
miDfSub01 <- miDf01[miDf01$numbers > 2, ]
print(miDfSub01)
```

```
##  numbers logicals characters
## 3      3     FALSE          c
## 4      4      TRUE          d
```

```
miDfSub02 <- miDf01[(miDf01$logicals == TRUE) & (miDf01$numbers < 2), ]
print(miDfSub02)
```

```
##  numbers logicals characters
## 1      1      TRUE          a
```

```
miDfSub03 <- miDf01[(miDf01$numbers %% 2) == 0, ]
print(miDfSub03)
```

```
##  numbers logicals characters
## 2      2      TRUE          b
## 4      4      TRUE          d
```

```
miDfSub04 <- miDf01[((miDf01$numbers %% 2) == 0) | (miDf01$logicals == TRUE), ]
print(miDfSub04)
```

```
##  numbers logicals characters
## 1      1      TRUE          a
## 2      2      TRUE          b
## 4      4      TRUE          d
```

### 8.3.3 Modificar un data.frame

Para agregar un elemento a un `data.frame`, procedemos como para un contenedor de tipo `list`. Es necesario asegurarse de que el nuevo elemento sea del mismo tamaño que los otros elementos de nuestro `data.frame`. Por defecto, un nuevo elemento en `data.frame` toma el nombre de la letra `V` seguido del número de la columna. Podemos cambiar los nombres de las columnas con la función `colnames()`. Podemos nombrar las líneas con la función `rownames()`.

```
newVec <- c(4, 5, 6, 7)
miDf01[[4]] <- newVec
print(miDf01)
```

```
##  numbers logicals characters V4
## 1      1      TRUE          a  4
## 2      2      TRUE          b  5
## 3      3     FALSE          c  6
## 4      4      TRUE          d  7
```

```
print(colnames(miDf01))
```

```
## [1] "numbers" "logicals" "characters" "V4"
```

```
colnames(miDf01)[4] <- "newVec"
print(miDf01)
```

```
##  numbers logicals characters newVec
## 1      1      TRUE          a      4
## 2      2      TRUE          b      5
## 3      3     FALSE          c      6
## 4      4      TRUE          d      7
```

```
print(rownames(miDf01))
```

```
## [1] "1" "2" "3" "4"
```

```
rownames(miDf01) <- c("row1", "row2", "row3", "row4")
print(miDf01)
```

```
##      numbers logicals characters newVec
## row1      1      TRUE          a      4
## row2      2      TRUE          b      5
## row3      3     FALSE          c      6
## row4      4      TRUE          d      7
```

```
newVec2 <- c(40, 50, 60, 70)
miDf01$newVec2 <- newVec2
print(miDf01)
```

```
##      numbers logicals characters newVec newVec2
## row1      1      TRUE          a      4      40
## row2      2      TRUE          b      5      50
## row3      3     FALSE          c      6      60
## row4      4      TRUE          d      7      70
```

Como el contenedor de tipo `data.frame` es un caso especial de `list`, la selección y modificación se realiza como un contenedor de tipo `list`. Dado que los elementos de un `data.frame` son del tipo vector, la selección y la modificación de los elementos de un `data.frame` se hace como para un contenedor vector.

```
miDf01$newVec2 <- miDf01$newVec2 * 2
print(miDf01)
```

```
##      numbers logicals characters newVec newVec2
## row1      1      TRUE          a      4      80
## row2      2      TRUE          b      5     100
## row3      3     FALSE          c      6     120
## row4      4      TRUE          d      7     140
```

```
miDf01$newVec2 + miDf01$newVec
```

```
## [1]  84 105 126 147
```

```
miDf01$newVec2[2] <- 0
print(miDf01)
```

```
##      numbers logicals characters newVec newVec2
## row1      1      TRUE          a      4      80
## row2      2      TRUE          b      5       0
## row3      3     FALSE          c      6     120
## row4      4      TRUE          d      7     140
```

Un vector se puede transformar en `data.frame` con la función `as.data.frame()`.

```
print(newVec2)
```

```
## [1] 40 50 60 70
```

```
print(as.data.frame(newVec2))
```

```
##   newVec2
## 1      40
## 2      50
## 3      60
## 4      70
```

```
is.data.frame(newVec2)
```

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

```
is.data.frame(as.data.frame(newVec2))
```

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

## 8.4 El contenedor `matrix`

El contenedor `matrix` se puede ver como un vector de dos dimensiones: líneas y columnas. Corresponde a una matriz en matemáticas, y puede contener solo un tipo de datos (`logical`, `numeric`, `character`, ...).

### 8.4.1 Crear una matrix

Para crear una matrix primero creamos un vector, luego especificamos el número deseado de líneas y columnas en la función `matrix()`.

```
vecForMatrix <- c(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12)
miMat <- matrix(vecForMatrix, nrow = 3, ncol = 4)
print(miMat)
```

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

No tenemos que especificar el número de líneas `nrow` y el número de columnas `ncol`. Si usamos uno u otro de estos argumentos, R calculará automáticamente el número correspondiente.

```
miMat <- matrix(vecForMatrix, nrow = 3)
print(miMat)
```

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

```
miMat <- matrix(vecForMatrix, ncol = 4)
print(miMat)
```

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

Observamos que los diferentes elementos del vector inicial aparecen por columna. Si queremos llenar la matrix empezando por línea, entonces tenemos que dar como valor `TRUE` al argumento `byrow`.

```
miMat <- matrix(vecForMatrix, nrow = 3, byrow = TRUE)
print(miMat)
```

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

También podemos dar un nombre a las líneas y columnas de nuestra matrix cuando se crea con el argumento `dimnames` que toma como valor una list de dos elementos: el nombre de las líneas y luego el nombre de las columnas. También podemos cambiar el nombre de las líneas y columnas a posteriori con las funciones `rownames()` y `colnames()`.

```
miMat <- matrix(
  vecForMatrix,
  nrow = 3,
  byrow = TRUE,
  dimnames = list(c("r1", "r2", "r3"), c("c1", "c2", "c3", "c4"))
```

```
)
print(miMat)
```

```
##      c1 c2 c3 c4
## r1   1  2  3  4
## r2   5  6  7  8
## r3   9 10 11 12
```

```
colnames(miMat) <- c("col1", "col2", "col3", "col4")
rownames(miMat) <- c("row1", "row2", "row3")
print(miMat)
```

```
##      col1 col2 col3 col4
## row1     1     2     3     4
## row2     5     6     7     8
## row3     9    10    11    12
```

Es posible crear una matrix desde un data.frame con la función `as.matrix()`. Tenemos que verificar que nuestra data.frame contenga solo elementos del mismo tipo (por ejemplo, elementos de tipo `numeric`).

```
vecForMat01 <- c(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12)
vecForMat02 <- vecForMat01 * 10
vecForMat03 <- vecForMat01 / 10
dfForMat <- data.frame(vecForMat01, vecForMat02, vecForMat03)
print(dfForMat)
```

```
##      vecForMat01 vecForMat02 vecForMat03
## 1              1           10          0.1
## 2              2           20          0.2
## 3              3           30          0.3
## 4              4           40          0.4
## 5              5           50          0.5
## 6              6           60          0.6
## 7              7           70          0.7
## 8              8           80          0.8
## 9              9           90          0.9
## 10             10          100          1.0
## 11             11          110          1.1
## 12             12          120          1.2
```

```
is.matrix(dfForMat)
```

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

```
as.matrix(dfForMat)
```

```
##      vecForMat01 vecForMat02 vecForMat03
## [1,]           1           10          0.1
## [2,]           2           20          0.2
## [3,]           3           30          0.3
## [4,]           4           40          0.4
## [5,]           5           50          0.5
## [6,]           6           60          0.6
```

```
## [7,]      7      70      0.7
## [8,]      8      80      0.8
## [9,]      9      90      0.9
## [10,]     10     100      1.0
## [11,]     11     110      1.1
## [12,]     12     120      1.2
```

```
is.matrix(as.matrix(dfForMat))
```

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

También podemos crear una matrix desde un vector con la función `as.matrix()` (matriz de una sola columna).

```
as.matrix(vecForMat01)
```

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

### 8.4.2 Manipular y hacer operaciones en una matrix

Todas las operaciones término a término son posibles con una matrix.

```
# operaciones término a término
miMat01 <- matrix(vecForMat01, ncol = 3)
miVecOp <- c(1, 10, 100, 1000)
miMat01 * miVecOp
```

```
##      [,1] [,2] [,3]
## [1,]    1    5    9
## [2,]   20   60  100
## [3,]  300  700 1100
## [4,] 4000 8000 12000
```

```
miMat01 + miVecOp
```

```
##      [,1] [,2] [,3]
## [1,]    2    6   10
## [2,]   12   16   20
## [3,]  103  107  111
## [4,] 1004 1008 1012
```

```
miMat01 / miMat01
```

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

```
miMat01 - 10
```

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

Para realizar operaciones algebraicas, podemos usar la función `%*%`.

```
# operaciones algebraicas
miVecConf <- c(1, 10, 100)
miMat01 %*% miVecConf
```

```
##      [,1]
## [1,]  951
## [2,] 1062
## [3,] 1173
## [4,] 1284
```

```
miMat02 <- matrix(c(1, 2, 3, 4, 5, 6, 7, 8, 9), ncol = 3)
print(miMat02)
```

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

```
miMat02 %*% miMat02
```

```
##      [,1] [,2] [,3]
## [1,]   30   66  102
## [2,]   36   81  126
## [3,]   42   96  150
```

La diagonal de una matrix se puede obtener con la función `diag()` y el determinante de una matrix con la función `det()`.

```
print(miMat02)
```

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



```
diag(miMat02)
```

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

```
det(miMat02)
```

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

Suele ser útil poder hacer una transposición de `matrix` (columnas en líneas o líneas en columnas). Para eso, están las funciones `aperm()` o `t()`. la función `t()` es más genérica y también funciona con `data.frame`.

```
aperm(miMat01)
```

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

```
t(miMat01)
```

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

### 8.4.3 Acceder a los elementos de una matrix

Tal como hemos hecho con los `data.frame`, podemos acceder a los elementos de una `matrix` especificando un número de línea y un número de columna entre corchetes simples `[ ]`, y separados por una coma. Si `i` es el número de línea y `j` es el número de columna, entonces `miMat01[i, j]` devuelve el elemento en la línea `i` y en la columna `j`. `miMat01[i, ]` devuelve todos los elementos de la línea `i`, y `miMat01[, j]` todos los elementos de la columna `j`. Múltiples selecciones son posibles. También podemos acceder a un elemento de acuerdo con su posición en la `matrix` entre corchetes simples `[ ]` contando por columna y luego por línea. En nuestro ejemplo, el valor del décimo elemento es 10.

```
i <- 2
j <- 1
print(miMat01[i, j])
```

```
## [1] 2
```

```
print(miMat01[i, ])
```

```
## [1] 2 6 10
```

```
print(miMat01[, j])
```

```
## [1] 1 2 3 4
```

```
print(miMat01[c(1, 2), c(2, 3)])
```

```
##      [,1] [,2]
## [1,]    5    9
## [2,]    6   10
```

```
print(miMat01[10])
```

```
## [1] 10
```

## 8.5 El contenedor array

El contenedor `array` es una generalización del contenedor de tipo `matrix`. Donde el tipo `matrix` tiene dos dimensiones (líneas y columnas), el tipo `array` tiene un número indefinido de dimensiones. Podemos saber el número de dimensiones de un `array` (y por lo tanto una `matrix`) con la función `dim()`.

```
dim(miMat01)
```

```
## [1] 4 3
```

### 8.5.1 Crear un array

La creación de una `array` es similar a la de una `matrix` con una dimensión extra.

```
miVecArr <- c(1, 2, 3, 4, 5, 6, 7, 8, 9)
miArray <- array(miVecArr, dim = c(3, 3, 2))
print(miArray)
```

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

```
dim(miArray)
```

```
## [1] 3 3 2
```

```
is.array(miArray)
```

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

```
miVecArr02 <- 10 * miVecArr
miArray02 <- array(c(miVecArr, miVecArr02), dim = c(3, 3, 2))
print(miArray02)
```

```
## , , 1
##
##      [,1] [,2] [,3]
## [1,]    1    4    7
## [2,]    2    5    8
## [3,]    3    6    9
##
## , , 2
##
##      [,1] [,2] [,3]
## [1,]   10   40   70
## [2,]   20   50   80
## [3,]   30   60   90
```

```
dim(miArray02)
```

```
## [1] 3 3 2
```

```
is.array(miArray02)
```

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

Podemos dar nombres a líneas y columnas, pero también a elementos.

```
miArray02 <- array(
  c(miVecArr, miVecArr02),
  dim = c(3, 3, 2),
  dimnames = list(
    c("r1", "r2", "r3"),
    c("c1", "c2", "c3"),
    c("matrix1", "matrix2")
  )
)
print(miArray02)
```

```
## , , matrix1
##
##      c1 c2 c3
## r1   1  4  7
## r2   2  5  8
## r3   3  6  9
##
## , , matrix2
##
##      c1 c2 c3
## r1  10 40 70
## r2  20 50 80
## r3  30 60 90
```

### 8.5.2 Manipular un array

La manipulación de un array se hace de la misma manera que para una `matrix`. Para acceder a los diferentes elementos de un array, simplemente hay que especificar la línea `i`, la columna `j`, y la `matrix` `k`.

```
i <- 2
j <- 1
k <- 1
print(miArray02[i, j, k])
```

```
## [1] 2
```

```
print(miArray02[, j, k])
```

```
## r1 r2 r3
##  1  2  3
```

```
print(miArray02[i, , k])
```

```
## c1 c2 c3
##  2  5  8
```

```
print(miArray02[i, j, ])
```

```
## matrix1 matrix2
##        2        20
```

## 8.6 Conclusión

Felicitaciones! Ahora conocemos los principales tipos de objetos que usaremos con R. Un objeto se caracteriza por sus atributos:

- el tipo de contenedor (`vector`, `data.frame`, `matrix`, `array`)
- el tipo de contenido de cada elemento (`numeric`, `logical`, `character`, ...)
- el valor de cada uno de los elementos (5, "qwerty", TRUE, ...)

Todos estos objetos se almacenan temporalmente en el entorno global de R (en la memoria RAM de nuestra computadora). El siguiente capítulo tratará las funciones y resaltaré uno de los aspectos que hace que R sea tan poderoso para analizar y administrar nuestros datos.

## Chapter 9

# Las funciones

### 9.1 ¿Qué es una función?

Con este capítulo vamos a echar un primer vistazo al poder de R a través de las funciones. Una función es un conjunto de líneas de código para realizar una tarea en particular. Hemos visto muchas funciones en capítulos anteriores, unas simples como la función `+` para añadir números, o otras más complejas como `c()` o `data.frame()` que permiten crear un `vector` o `data.frame`. En cualquier caso, se puede reconocer una función gracias a los paréntesis que la siguen en los cuales vamos a ingresar **argumentos**. Los argumentos corresponden a la información que queremos transmitir a nuestra función para que realice la tarea que queremos lograr.

Para funciones simples como `+`, los paréntesis han sido eliminados para que el código sea más fácil de leer, pero es una función que puede usarse con paréntesis si usamos el signo `+` entre comillas. Los argumentos son los números que queremos agregar.

```
5 + 2
```

```
## [1] 7
```

```
'+'(5, 2)
```

```
## [1] 7
```

En este capítulo nos enfocaremos en las funciones más comunes. No se trata de aprender todo de memoria, sino de saber que existen estas funciones y de poder consultar más adelante este capítulo como referencia. ¡Con tiempo y práctica eventualmente los sabremos de memoria! Hay más de 1000 funciones en la versión básica de R, y más de 10000 paquetes adicionales que se pueden instalar, cada uno con docenas de funciones. Antes de comenzar a escribir una nueva función, siempre debemos verificar que ya no exista.

### 9.2 Las funciones más comunes

Para trabajar con las funciones, vamos a usar los datos `iris` que están incluidos con la versión básica de R y que corresponden a la longitud y el ancho de los sépalos y pétalos de diferentes especies de iris. Los datos `iris` están en una `data.frame` de 5 columnas y 150 líneas. Para obtener más información sobre los datos `iris`, podemos consultar la documentación R con la función `help(iris)`. El acceso a la documentación es el tema de la siguiente sección.

## 9.2.1 El acceso a la documentación

### 9.2.1.1 help()

La función esencial de R es acceder a la documentación (en inglés). **Todas las funciones R tienen documentación.** Podemos acceder a la documentación con la función `help()` o usando el atajo `?`.

```
help(matrix) # equivalente a ?matrix
```

La documentación siempre está estructurada de la misma manera. Primero tenemos el nombre de la función buscada `matrix`, seguida entre llaves por el nombre del paquete R cuya función depende. Veremos cómo instalar paquetes adicionales más adelante. Por ahora tenemos los que vienen con la versión básica de R. Aquí podemos ver que la función `matrix()` depende del paquete `base`.

Podemos ver la etiqueta de la función (Matrices), seguida de los párrafos Description, Usage, y Arguments. Algunas veces se agregan los párrafos Details, Note, References y See also. El último párrafo es Ejemplos. La última línea de la documentación permite volver al índice del paquete del que depende la función consultada.

Al copiar `help(matrix)` en nuestra consola R, podemos ver que el párrafo Description indica lo que hace la función. En el caso de `help(matrix)`, hay tres funciones: `matrix()`, `as.matrix()` y `is.matrix()`.

```
# Description
# matrix creates a matrix from the given set of values.
# as.matrix attempts to turn its argument into a matrix.
# is.matrix tests if its argument is a (strict) matrix.
```

El párrafo Usage explica cómo usar la función y cuáles son los valores predeterminados para cada parámetro.

```
# Usage
# matrix(data = NA, nrow = 1, ncol = 1, byrow = FALSE,
#        dimnames = NULL)
```

La función `matrix()` puede tomar 5 argumentos: `data`, `nrow`, `ncol`, `byrow`, y `dimnames`. Podemos ver que por defecto una `matrix` consistirá de una sola línea y una sola columna, y que la información se completará por columna.

El párrafo Arguments detalla los valores y el tipo de contenedor de cada argumento de nuestra función. Por ejemplo, podemos ver que el argumento `dimnames` debe ser del tipo `list`. Es por eso que hemos usado este formato en la sección `matrix`.

```
# Arguments
# data      an optional data vector (including a list or expression vector).
#           Non-atomic classed R objects are coerced by as.vector and all
#           attributes discarded.
# nrow      the desired number of rows.
# ncol      the desired number of columns.
# byrow     logical. If FALSE (the default) the matrix is filled by columns,
#           otherwise the matrix is filled by rows.
# dimnames  A dimnames attribute for the matrix: NULL or a list of length 2
#           giving the row and column names respectively. An empty list is
#           treated as NULL, and a list of length one as row names. The
#           list can be named, and the list names will be used as names for
#           the dimensions.
```

El párrafo Details proporciona elementos adicionales en la función. El párrafo Examples proporciona ejemplos reproducibles en la consola.

```
## Example of setting row and column names
mdat <- matrix(c(1,2,3, 11,12,13), nrow = 2, ncol = 3, byrow = TRUE,
               dimnames = list(c("row1", "row2"),
                               c("C.1", "C.2", "C.3")))
mdat
```

```
##      C.1 C.2 C.3
## row1  1  2  3
## row2 11 12 13
```

El nombre de los argumentos no es necesario para que una función sea interpretada correctamente por R. Sin embargo, es mejor usar explícitamente el nombre de los argumentos seguidos por el signo = para que el código sea más legible.

```
# buen ejemplo
mdat <- matrix(c(1, 2, 3, 11, 12, 13), nrow = 2, ncol = 3, byrow = TRUE)
# mal ejemplo
mdat <- matrix(c(1, 2, 3, 11, 12, 13), 2, 3, TRUE)
```

### 9.2.1.2 help.search()

La función `help.search()` o `??` permite buscar una expresión en toda la documentación. Es útil cuando buscamos una función sin saber el nombre exacto de la función en R.

```
help.search("average")
```

La función `help.search()` devuelve una página que contiene la lista de páginas donde se encontró la expresión en la forma `package-name::function-name`.

## 9.2.2 Ver los datos

### 9.2.2.1 str()

La función `str()` permite visualizar la estructura interna de un objeto, como se indica en la documentación que podemos consultar con `help(str)`.

```
str(iris)
```

```
## 'data.frame':   150 obs. of  5 variables:
## $ Sepal.Length: num  5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
## $ Sepal.Width : num  3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...
## $ Petal.Length: num  1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ...
## $ Petal.Width : num  0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ...
## $ Species     : Factor w/ 3 levels "setosa","versicolor",...: 1 1 1 1 1 1 1 1 1 1 ...
```

La función `str()` devuelve el tipo de objeto (`data.frame`), el número de observaciones (150), el número de variables (5), el nombre de cada variable (`Sepal.Length`, `Sepal.Width`, `Petal.Length`, `Petal.Width`, y `Species`), el tipo de cada variable (`num`, `Factor`), y los primeros valores de cada una de las variables. Es una función útil para echar un vistazo a un conjunto de datos, pero también para verificar que los datos sean del tipo requerido antes de realizar un análisis estadístico.

### 9.2.2.2 head() y tail()

La función `head()` devuelve los primeros valores de un objeto, y la función `tail()` devuelve los últimos valores de un objeto. Por defecto, se devuelven seis valores, el argumento `n` controla el número de valores a devolver.

```
head(iris)
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 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
```

```
tail(iris)
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width  Species
## 145         6.7         3.3         5.7         2.5 virginica
## 146         6.7         3.0         5.2         2.3 virginica
## 147         6.3         2.5         5.0         1.9 virginica
## 148         6.5         3.0         5.2         2.0 virginica
## 149         6.2         3.4         5.4         2.3 virginica
## 150         5.9         3.0         5.1         1.8 virginica
```

```
head(iris, n = 2)
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1         5.1         3.5         1.4         0.2   setosa
## 2         4.9         3.0         1.4         0.2   setosa
```

### 9.2.2.3 names()

Ya hemos visto la función `names()`, que permite conocer los nombres de los elementos de un objeto, pero también asignar nombres a los elementos de un objeto como a un `matrix`, a una `list` o a un `data.frame`.

```
names(iris)
```

```
## [1] "Sepal.Length" "Sepal.Width"  "Petal.Length" "Petal.Width"
## [5] "Species"
```

```
irisCopy <- iris
names(irisCopy) <- c("a", "b", "c", "d", "e")
names(irisCopy)
```

```
## [1] "a" "b" "c" "d" "e"
```

### 9.2.2.4 cat() et print()

La función `cat()` se usa para mostrar el contenido de un objeto mientras que la función `print()` devuelve el valor de un objeto con la capacidad de realizar conversiones.



```
cat(names(iris))
```

```
## Sepal.Length Sepal.Width Petal.Length Petal.Width Species
```

```
print(names(iris))
```

```
## [1] "Sepal.Length" "Sepal.Width"  "Petal.Length" "Petal.Width"
## [5] "Species"
```

```
cat(iris[1, 1])
```

```
## 5.1
```

```
print(iris[1, 1])
```

```
## [1] 5.1
```

```
print(iris[1, 1], digits = 0)
```

```
## [1] 5
```

### 9.2.3 Manipular los datos

#### 9.2.3.1 rank()

La función `rank()` devuelve el número de la posición ordenada de cada elemento de un conjunto de elementos. En el caso de elementos del mismo valor, el argumento `ties.method` hace posible hacer una elección sobre la clasificación. Como con todas las funciones, los detalles están presentes en la documentación.

```
vecManip <- c(10, 20, 30, 70, 60, 50, 40)
rank(vecManip)
```

```
## [1] 1 2 3 7 6 5 4
```

```
vecManip2 <- c(10, 20, 30, 10, 50, 10, 40)
rank(vecManip2)
```

```
## [1] 2 4 5 2 7 2 6
```

```
rank(vecManip2, ties.method = "first")
```

```
## [1] 1 4 5 2 7 3 6
```

```
rank(vecManip2, ties.method = "min")
```

```
## [1] 1 4 5 1 7 1 6
```

```
print(iris[, 1])
```

```
## [1] 5.1 4.9 4.7 4.6 5.0 5.4 4.6 5.0 4.4 4.9 5.4 4.8 4.8 4.3 5.8 5.7 5.4
## [18] 5.1 5.7 5.1 5.4 5.1 4.6 5.1 4.8 5.0 5.0 5.2 5.2 4.7 4.8 5.4 5.2 5.5
## [35] 4.9 5.0 5.5 4.9 4.4 5.1 5.0 4.5 4.4 5.0 5.1 4.8 5.1 4.6 5.3 5.0 7.0
## [52] 6.4 6.9 5.5 6.5 5.7 6.3 4.9 6.6 5.2 5.0 5.9 6.0 6.1 5.6 6.7 5.6 5.8
## [69] 6.2 5.6 5.9 6.1 6.3 6.1 6.4 6.6 6.8 6.7 6.0 5.7 5.5 5.5 5.8 6.0 5.4
## [86] 6.0 6.7 6.3 5.6 5.5 5.5 6.1 5.8 5.0 5.6 5.7 5.7 6.2 5.1 5.7 6.3 5.8
## [103] 7.1 6.3 6.5 7.6 4.9 7.3 6.7 7.2 6.5 6.4 6.8 5.7 5.8 6.4 6.5 7.7 7.7
## [120] 6.0 6.9 5.6 7.7 6.3 6.7 7.2 6.2 6.1 6.4 7.2 7.4 7.9 6.4 6.3 6.1 7.7
## [137] 6.3 6.4 6.0 6.9 6.7 6.9 5.8 6.8 6.7 6.7 6.3 6.5 6.2 5.9
```

```
rank(iris[, 1], ties.method = "average")
```

```
## [1] 37.0 19.5 10.5 7.5 27.5 49.5 7.5 27.5 3.0 19.5 49.5
## [12] 14.0 14.0 1.0 77.0 69.5 49.5 37.0 69.5 37.0 49.5 37.0
## [23] 7.5 37.0 14.0 27.5 27.5 43.5 43.5 10.5 14.0 49.5 43.5
## [34] 56.0 19.5 27.5 56.0 19.5 3.0 37.0 27.5 5.0 3.0 27.5
## [45] 37.0 14.0 37.0 7.5 46.0 27.5 138.0 112.0 135.5 56.0 118.0
## [56] 69.5 104.0 19.5 121.5 43.5 27.5 82.0 86.5 92.5 62.5 126.5
## [67] 62.5 77.0 97.5 62.5 82.0 92.5 104.0 92.5 112.0 121.5 132.0
## [78] 126.5 86.5 69.5 56.0 56.0 77.0 86.5 49.5 86.5 126.5 104.0
## [89] 62.5 56.0 56.0 92.5 77.0 27.5 62.5 69.5 69.5 97.5 37.0
## [100] 69.5 104.0 77.0 139.0 104.0 118.0 145.0 19.5 143.0 126.5 141.0
## [111] 118.0 112.0 132.0 69.5 77.0 112.0 118.0 147.5 147.5 86.5 135.5
## [122] 62.5 147.5 104.0 126.5 141.0 97.5 92.5 112.0 141.0 144.0 150.0
## [133] 112.0 104.0 92.5 147.5 104.0 112.0 86.5 135.5 126.5 135.5 77.0
## [144] 132.0 126.5 126.5 104.0 118.0 97.5 82.0
```

```
# help(rank)
# ...
# Usage
# rank(x, na.last = TRUE,
#      ties.method = c("average", "first", "last",
#                      "random", "max", "min"))
```

### 9.2.3.2 order()

La función `order()` devuelve el número de la reorganización de los elementos en función de su posición. Es muy útil, por ejemplo, para ordenar un `data.frame` en función de una columna.

```
print(vecManip2)
```

```
## [1] 10 20 30 10 50 10 40
```

```
rank(vecManip2)
```

```
## [1] 2 4 5 2 7 2 6
```

```
order(vecManip2)
```

```
## [1] 1 4 6 2 3 7 5
```

```
print(iris[, 1])
```

```
##      [1] 5.1 4.9 4.7 4.6 5.0 5.4 4.6 5.0 4.4 4.9 5.4 4.8 4.8 4.3 5.8 5.7 5.4
##     [18] 5.1 5.7 5.1 5.4 5.1 4.6 5.1 4.8 5.0 5.0 5.2 5.2 4.7 4.8 5.4 5.2 5.5
##     [35] 4.9 5.0 5.5 4.9 4.4 5.1 5.0 4.5 4.4 5.0 5.1 4.8 5.1 4.6 5.3 5.0 7.0
##     [52] 6.4 6.9 5.5 6.5 5.7 6.3 4.9 6.6 5.2 5.0 5.9 6.0 6.1 5.6 6.7 5.6 5.8
##     [69] 6.2 5.6 5.9 6.1 6.3 6.1 6.4 6.6 6.8 6.7 6.0 5.7 5.5 5.5 5.8 6.0 5.4
##     [86] 6.0 6.7 6.3 5.6 5.5 5.5 6.1 5.8 5.0 5.6 5.7 5.7 6.2 5.1 5.7 6.3 5.8
##    [103] 7.1 6.3 6.5 7.6 4.9 7.3 6.7 7.2 6.5 6.4 6.8 5.7 5.8 6.4 6.5 7.7 7.7
##    [120] 6.0 6.9 5.6 7.7 6.3 6.7 7.2 6.2 6.1 6.4 7.2 7.4 7.9 6.4 6.3 6.1 7.7
##    [137] 6.3 6.4 6.0 6.9 6.7 6.9 5.8 6.8 6.7 6.7 6.3 6.5 6.2 5.9
```

```
rank(iris[, 1])
```

```
##      [1] 37.0 19.5 10.5 7.5 27.5 49.5 7.5 27.5 3.0 19.5 49.5
##     [12] 14.0 14.0 1.0 77.0 69.5 49.5 37.0 69.5 37.0 49.5 37.0
##     [23] 7.5 37.0 14.0 27.5 27.5 43.5 43.5 10.5 14.0 49.5 43.5
##     [34] 56.0 19.5 27.5 56.0 19.5 3.0 37.0 27.5 5.0 3.0 27.5
##     [45] 37.0 14.0 37.0 7.5 46.0 27.5 138.0 112.0 135.5 56.0 118.0
##     [56] 69.5 104.0 19.5 121.5 43.5 27.5 82.0 86.5 92.5 62.5 126.5
##     [67] 62.5 77.0 97.5 62.5 82.0 92.5 104.0 92.5 112.0 121.5 132.0
##     [78] 126.5 86.5 69.5 56.0 56.0 77.0 86.5 49.5 86.5 126.5 104.0
##     [89] 62.5 56.0 56.0 92.5 77.0 27.5 62.5 69.5 69.5 97.5 37.0
##    [100] 69.5 104.0 77.0 139.0 104.0 118.0 145.0 19.5 143.0 126.5 141.0
##    [111] 118.0 112.0 132.0 69.5 77.0 112.0 118.0 147.5 147.5 86.5 135.5
##    [122] 62.5 147.5 104.0 126.5 141.0 97.5 92.5 112.0 141.0 144.0 150.0
##    [133] 112.0 104.0 92.5 147.5 104.0 112.0 86.5 135.5 126.5 135.5 77.0
##    [144] 132.0 126.5 126.5 104.0 118.0 97.5 82.0
```

```
order(iris[, 1])
```

```
##      [1] 14 9 39 43 42 4 7 23 48 3 30 12 13 25 31 46 2
##     [18] 10 35 38 58 107 5 8 26 27 36 41 44 50 61 94 1 18
##     [35] 20 22 24 40 45 47 99 28 29 33 60 49 6 11 17 21 32
##     [52] 85 34 37 54 81 82 90 91 65 67 70 89 95 122 16 19 56
##     [69] 80 96 97 100 114 15 68 83 93 102 115 143 62 71 150 63 79
##     [86] 84 86 120 139 64 72 74 92 128 135 69 98 127 149 57 73 88
##    [103] 101 104 124 134 137 147 52 75 112 116 129 133 138 55 105 111 117
##    [120] 148 59 76 66 78 87 109 125 141 145 146 77 113 144 53 121 140
##    [137] 142 51 103 110 126 130 108 131 106 118 119 123 136 132
```

```
head(iris[order(iris[, 1]),], n = 10)
```

```
##      Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 14              4.3          3.0           1.1          0.1  setosa
## 9               4.4          2.9           1.4          0.2  setosa
## 39              4.4          3.0           1.3          0.2  setosa
## 43              4.4          3.2           1.3          0.2  setosa
## 42              4.5          2.3           1.3          0.3  setosa
## 4               4.6          3.1           1.5          0.2  setosa
## 7               4.6          3.4           1.4          0.3  setosa
## 23              4.6          3.6           1.0          0.2  setosa
```

```
## 48      4.6      3.2      1.4      0.2  setosa
## 3       4.7      3.2      1.3      0.2  setosa
```

### 9.2.3.3 sort()

La función `sort()` se usa para ordenar los elementos de un objeto. No permite la clasificación por más de una variable, como es el caso de `order()`.

```
print(vecManip2)
```

```
## [1] 10 20 30 10 50 10 40
```

```
sort(vecManip2)
```

```
## [1] 10 10 10 20 30 40 50
```

```
print(iris[, 1])
```

```
## [1] 5.1 4.9 4.7 4.6 5.0 5.4 4.6 5.0 4.4 4.9 5.4 4.8 4.8 4.3 5.8 5.7 5.4
## [18] 5.1 5.7 5.1 5.4 5.1 4.6 5.1 4.8 5.0 5.0 5.2 5.2 4.7 4.8 5.4 5.2 5.5
## [35] 4.9 5.0 5.5 4.9 4.4 5.1 5.0 4.5 4.4 5.0 5.1 4.8 5.1 4.6 5.3 5.0 7.0
## [52] 6.4 6.9 5.5 6.5 5.7 6.3 4.9 6.6 5.2 5.0 5.9 6.0 6.1 5.6 6.7 5.6 5.8
## [69] 6.2 5.6 5.9 6.1 6.3 6.1 6.4 6.6 6.8 6.7 6.0 5.7 5.5 5.5 5.8 6.0 5.4
## [86] 6.0 6.7 6.3 5.6 5.5 5.5 6.1 5.8 5.0 5.6 5.7 5.7 6.2 5.1 5.7 6.3 5.8
## [103] 7.1 6.3 6.5 7.6 4.9 7.3 6.7 7.2 6.5 6.4 6.8 5.7 5.8 6.4 6.5 7.7 7.7
## [120] 6.0 6.9 5.6 7.7 6.3 6.7 7.2 6.2 6.1 6.4 7.2 7.4 7.9 6.4 6.3 6.1 7.7
## [137] 6.3 6.4 6.0 6.9 6.7 6.9 5.8 6.8 6.7 6.7 6.3 6.5 6.2 5.9
```

```
sort(iris[, 1])
```

```
## [1] 4.3 4.4 4.4 4.4 4.5 4.6 4.6 4.6 4.6 4.7 4.7 4.8 4.8 4.8 4.8 4.8 4.9
## [18] 4.9 4.9 4.9 4.9 4.9 5.0 5.0 5.0 5.0 5.0 5.0 5.0 5.0 5.0 5.0 5.1 5.1
## [35] 5.1 5.1 5.1 5.1 5.1 5.1 5.1 5.2 5.2 5.2 5.2 5.3 5.4 5.4 5.4 5.4 5.4
## [52] 5.4 5.5 5.5 5.5 5.5 5.5 5.5 5.5 5.6 5.6 5.6 5.6 5.6 5.6 5.7 5.7 5.7
## [69] 5.7 5.7 5.7 5.7 5.7 5.8 5.8 5.8 5.8 5.8 5.8 5.8 5.9 5.9 5.9 6.0 6.0
## [86] 6.0 6.0 6.0 6.0 6.1 6.1 6.1 6.1 6.1 6.1 6.2 6.2 6.2 6.2 6.3 6.3 6.3
## [103] 6.3 6.3 6.3 6.3 6.3 6.3 6.4 6.4 6.4 6.4 6.4 6.4 6.4 6.5 6.5 6.5 6.5
## [120] 6.5 6.6 6.6 6.7 6.7 6.7 6.7 6.7 6.7 6.7 6.8 6.8 6.8 6.9 6.9 6.9
## [137] 6.9 7.0 7.1 7.2 7.2 7.2 7.3 7.4 7.6 7.7 7.7 7.7 7.7 7.7 7.9
```

### 9.2.3.4 append()

Esta función se usa para agregar un elemento a un vector en una posición determinada por el argumento `after`. Esta función también es más rápida que su alternativa `c()`.

```
print(vecManip2)
```

```
## [1] 10 20 30 10 50 10 40
```

```
append(vecManip2, 5)
```

```
## [1] 10 20 30 10 50 10 40 5
```

```
append(vecManip2, 5, after = 2)
```

```
## [1] 10 20 5 30 10 50 10 40
```

### 9.2.3.5 cbind() et rbind()

Las funciones `cbind()` y `rbind()` permiten combinar elementos por columna o por línea.

```
cbind(vecManip2, vecManip2)
```

```
##      vecManip2 vecManip2
## [1,]        10        10
## [2,]        20        20
## [3,]        30        30
## [4,]        10        10
## [5,]        50        50
## [6,]        10        10
## [7,]        40        40
```

```
rbind(vecManip2, vecManip2)
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7]
## vecManip2 10  20  30  10  50  10  40
## vecManip2 10  20  30  10  50  10  40
```

### 9.2.3.6 paste() et paste0()

Estas son dos funciones que usaremos mucho a partir de ahora. Las funciones `paste()` y `paste0()` se usan para concatenar cadenas de texto. La función `paste0()` es equivalente a `paste()` sin proponer un separador entre los elementos a concatenar. La función `paste0()` también es más rápida.

```
paste(1, "a")
```

```
## [1] "1 a"
```

```
paste0(1, "a")
```

```
## [1] "1a"
```

```
paste(1, "a", sep = "_")
```

```
## [1] "1_a"
```

```
paste0("prefix_", vecManip2, "_suffix")
```

```
## [1] "prefix_10_suffix" "prefix_20_suffix" "prefix_30_suffix"
## [4] "prefix_10_suffix" "prefix_50_suffix" "prefix_10_suffix"
## [7] "prefix_40_suffix"
```

```
paste(vecManip2, rank(vecManip2), sep = "_")
```

```
## [1] "10_2" "20_4" "30_5" "10_2" "50_7" "10_2" "40_6"
```

### 9.2.3.7 rev()

La función `rev()` devuelve los elementos de un objeto en orden inverso.

```
print(vecManip2)
```

```
## [1] 10 20 30 10 50 10 40
```

```
rev(vecManip2)
```

```
## [1] 40 10 50 10 30 20 10
```

### 9.2.3.8 %in%()

La función `%in%()` se puede comparar con un operador de comparación. Esta función toma dos objetos como argumentos y devuelve TRUE o FALSE para cada elemento del primer objeto de acuerdo con su presencia o ausencia en el segundo objeto. Para acceder a la documentación de la función, use `help('%in%')` (con comillas simples).

```
print(vecManip)
```

```
## [1] 10 20 30 70 60 50 40
```

```
print(vecManip2)
```

```
## [1] 10 20 30 10 50 10 40
```

```
vecManip %in% vecManip2
```

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

```
vecManip2 %in% vecManip
```

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

## 9.2.4 Funciones matemáticas

Ya hemos visto las funciones `+`, `-`, `*`, `/`, `^`, `%%` y otros operadores aritméticos. R también tiene funciones matemáticas básicas como exponencial `exp()`, raíz cuadrada `sqrt()`, valor absoluto `abs()`, sinus `sin()`, coseno `cos()`, tangente `tan()`, logaritmo `log()`, logaritmo base 10 `log10()`, arco coseno `acos()`, arco sinus `asin()`, y arco tangente `atan()`.

```
print(vecManip2)
```

```
## [1] 10 20 30 10 50 10 40
```

```
exp(vecManip2)
```

```
## [1] 2.202647e+04 4.851652e+08 1.068647e+13 2.202647e+04 5.184706e+21  
## [6] 2.202647e+04 2.353853e+17
```

```
sqrt(vecManip2)
```

```
## [1] 3.162278 4.472136 5.477226 3.162278 7.071068 3.162278 6.324555
```

```
abs(-vecManip2)
```

```
## [1] 10 20 30 10 50 10 40
```

```
sin(vecManip2)
```

```
## [1] -0.5440211 0.9129453 -0.9880316 -0.5440211 -0.2623749 -0.5440211  
## [7] 0.7451132
```

```
cos(vecManip2)
```

```
## [1] -0.8390715 0.4080821 0.1542514 -0.8390715 0.9649660 -0.8390715  
## [7] -0.6669381
```

```
tan(vecManip2)
```

```
## [1] 0.6483608 2.2371609 -6.4053312 0.6483608 -0.2719006 0.6483608  
## [7] -1.1172149
```

```
log(vecManip2)
```

```
## [1] 2.302585 2.995732 3.401197 2.302585 3.912023 2.302585 3.688879
```

```
log10(vecManip2)
```

```
## [1] 1.000000 1.301030 1.477121 1.000000 1.698970 1.000000 1.602060
```

```
acos(vecManip2/100)
```

```
## [1] 1.470629 1.369438 1.266104 1.470629 1.047198 1.470629 1.159279
```

```
asin(vecManip2/100)
```

```
## [1] 0.1001674 0.2013579 0.3046927 0.1001674 0.5235988 0.1001674 0.4115168
```

```
atan(vecManip2/100)
```

```
## [1] 0.09966865 0.19739556 0.29145679 0.09966865 0.46364761 0.09966865  
## [7] 0.38050638
```

### 9.2.5 Estadísticas descriptivas

También podemos realizar estadísticas descriptivas de forma muy simple a partir de un conjunto de datos.

#### 9.2.5.1 `mean()`

La función `mean()` devuelve la media. Para ignorar los valores faltantes NA, hay que afectar el valor `TRUE` al argumento `na.rm()`.

```
mean(iris[, 1])
```

```
## [1] 5.843333
```

```
vecManip3 <- c(1, 5, 6, 8, NA, 45, NA, 14)  
mean(vecManip3)
```

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

```
mean(vecManip3, na.rm = TRUE)
```

```
## [1] 13.16667
```

#### 9.2.5.2 `sd()`

La función `sd()` devuelve la desviación estándar.

```
sd(iris[, 1])
```

```
## [1] 0.8280661
```

```
print(vecManip3)
```

```
## [1] 1 5 6 8 NA 45 NA 14
```

```
sd(vecManip3)
```

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

```
sd(vecManip3, na.rm = TRUE)
```

```
## [1] 16.16684
```

#### 9.2.5.3 `max()` y `min()`

La función `max()` devuelve el valor máximo y `min()` el valor mínimo.

```
max(iris[, 1])
```

```
## [1] 7.9
```



```
print(vecManip3)
```

```
## [1] 1 5 6 8 NA 45 NA 14
```

```
max(vecManip3)
```

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

```
max(vecManip3, na.rm = TRUE)
```

```
## [1] 45
```

```
min(iris[, 1])
```

```
## [1] 4.3
```

```
min(vecManip3)
```

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

```
min(vecManip3, na.rm = TRUE)
```

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

#### 9.2.5.4 quantile()

La función `quantile()` devuelve el cuantil definido por el argumento `probs`.

```
quantile(iris[, 1])
```

```
## 0% 25% 50% 75% 100%
## 4.3 5.1 5.8 6.4 7.9
```

```
quantile(iris[, 1], probs = c(0, 0.25, 0.5, 0.75, 1))
```

```
## 0% 25% 50% 75% 100%
## 4.3 5.1 5.8 6.4 7.9
```

```
quantile(iris[, 1], probs = c(0, 0.1, 0.5, 0.9, 1))
```

```
## 0% 10% 50% 90% 100%
## 4.3 4.8 5.8 6.9 7.9
```

#### 9.2.5.5 summary()

La función `summary()` devuelve un resumen con el mínimo, primer cuartil, mediana, promedio, tercer cuartil y máximo.

```
summary(iris[, 1])
```

```
##      Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##    4.300   5.100   5.800   5.843   6.400   7.900
```

#### 9.2.5.6 median()

La función `median()` devuelve la mediana.

```
median(iris[, 1])
```

```
## [1] 5.8
```

```
print(vecManip3)
```

```
## [1]  1  5  6  8 NA 45 NA 14
```

```
median(vecManip3)
```

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

```
median(vecManip3, na.rm = TRUE)
```

```
## [1] 7
```

#### 9.2.5.7 length()

La función `length()` devuelve el tamaño de un objeto (número de elementos).

```
length(iris[, 1])
```

```
## [1] 150
```

```
print(vecManip3)
```

```
## [1]  1  5  6  8 NA 45 NA 14
```

```
length(vecManip3)
```

```
## [1] 8
```

#### 9.2.5.8 nrow() et ncol()

La función `nrow()` devuelve el número de líneas y la función `ncol()` el número de columnas en un objeto.

```
nrow(iris)
```

```
## [1] 150
```

```
ncol(iris)
```

```
## [1] 5
```

**9.2.5.9 round(), ceiling(), floor(), et trunc()**

La función `round()` le permite seleccionar una cierta cantidad de decimales (0 por defecto)

```
round(5.56874258564)
```

```
## [1] 6
```

```
round(5.56874258564, digits = 2)
```

```
## [1] 5.57
```

La función `ceiling()` devuelve el entero más pequeño que no es inferior al valor especificado.

```
ceiling(5.9999)
```

```
## [1] 6
```

```
ceiling(5.0001)
```

```
## [1] 6
```

La función `floor()` devuelve el entero más grande que no excede el valor especificado.

```
floor(5.9999)
```

```
## [1] 5
```

```
floor(5.0001)
```

```
## [1] 5
```

La función `trunc()` devuelve la parte entera del valor especificado.

```
trunc(5.9999)
```

```
## [1] 5
```

```
trunc(5.0001)
```

```
## [1] 5
```

**9.2.5.10 rowSums() et colSums()**

Las funciones `rowSums()` y `colSums()` calculan la suma de filas y columnas.

```
rowSums(iris[, c(1, 2, 3, 4)])
```

```
## [1] 10.2  9.5  9.4  9.4 10.2 11.4  9.7 10.1  8.9  9.6 10.8 10.0  9.3  8.5
## [15] 11.2 12.0 11.0 10.3 11.5 10.7 10.7 10.7  9.4 10.6 10.3  9.8 10.4 10.4
## [29] 10.2  9.7  9.7 10.7 10.9 11.3  9.7  9.6 10.5 10.0  8.9 10.2 10.1  8.4
## [43]  9.1 10.7 11.2  9.5 10.7  9.4 10.7  9.9 16.3 15.6 16.4 13.1 15.4 14.3
```

```
## [57] 15.9 11.6 15.4 13.2 11.5 14.6 13.2 15.1 13.4 15.6 14.6 13.6 14.4 13.1
## [71] 15.7 14.2 15.2 14.8 14.9 15.4 15.8 16.4 14.9 12.8 12.8 12.6 13.6 15.4
## [85] 14.4 15.5 16.0 14.3 14.0 13.3 13.7 15.1 13.6 11.6 13.8 14.1 14.1 14.7
## [99] 11.7 13.9 18.1 15.5 18.1 16.6 17.5 19.3 13.6 18.3 16.8 19.4 16.8 16.3
## [113] 17.4 15.2 16.1 17.2 16.8 20.4 19.5 14.7 18.1 15.3 19.2 15.7 17.8 18.2
## [127] 15.6 15.8 16.9 17.6 18.2 20.1 17.0 15.7 15.7 19.1 17.7 16.8 15.6 17.5
## [141] 17.8 17.4 15.5 18.2 18.2 17.2 15.7 16.7 17.3 15.8
```

```
colSums(iris[, c(1, 2, 3, 4)])
```

```
## Sepal.Length Sepal.Width Petal.Length Petal.Width
##          876.5          458.6          563.7          179.9
```

### 9.2.5.11 rowMeans() et colMeans()

Las funciones `rowMeans()` y `colMeans()` calculan el promedio de filas y columnas.

```
rowMeans(iris[, c(1, 2, 3, 4)])
```

```
## [1] 2.550 2.375 2.350 2.350 2.550 2.850 2.425 2.525 2.225 2.400 2.700
## [12] 2.500 2.325 2.125 2.800 3.000 2.750 2.575 2.875 2.675 2.675 2.675
## [23] 2.350 2.650 2.575 2.450 2.600 2.600 2.550 2.425 2.425 2.675 2.725
## [34] 2.825 2.425 2.400 2.625 2.500 2.225 2.550 2.525 2.100 2.275 2.675
## [45] 2.800 2.375 2.675 2.350 2.675 2.475 4.075 3.900 4.100 3.275 3.850
## [56] 3.575 3.975 2.900 3.850 3.300 2.875 3.650 3.300 3.775 3.350 3.900
## [67] 3.650 3.400 3.600 3.275 3.925 3.550 3.800 3.700 3.725 3.850 3.950
## [78] 4.100 3.725 3.200 3.200 3.150 3.400 3.850 3.600 3.875 4.000 3.575
## [89] 3.500 3.325 3.425 3.775 3.400 2.900 3.450 3.525 3.525 3.675 2.925
## [100] 3.475 4.525 3.875 4.525 4.150 4.375 4.825 3.400 4.575 4.200 4.850
## [111] 4.200 4.075 4.350 3.800 4.025 4.300 4.200 5.100 4.875 3.675 4.525
## [122] 3.825 4.800 3.925 4.450 4.550 3.900 3.950 4.225 4.400 4.550 5.025
## [133] 4.250 3.925 3.925 4.775 4.425 4.200 3.900 4.375 4.450 4.350 3.875
## [144] 4.550 4.550 4.300 3.925 4.175 4.325 3.950
```

```
colMeans(iris[, c(1, 2, 3, 4)])
```

```
## Sepal.Length Sepal.Width Petal.Length Petal.Width
##    5.843333    3.057333    3.758000    1.199333
```

### 9.2.5.12 aggregate()

La función `aggregate()` permite agrupar los elementos de un objeto de acuerdo con un valor. El argumento `by` define el elemento sobre el que se realiza la agrupación. Debe ser del tipo `list`.

```
aggregate(iris[, c(1, 2, 3, 4)], by = list(iris$Species), FUN = mean)
```

```
##      Group.1 Sepal.Length Sepal.Width Petal.Length Petal.Width
## 1      setosa      5.006      3.428      1.462      0.246
## 2 versicolor      5.936      2.770      4.260      1.326
## 3 virginica      6.588      2.974      5.552      2.026
```

```
aggregate(iris[, c(1, 2)], by = list(iris$Species), FUN = summary)
```

```
##      Group.1 Sepal.Length.Min. Sepal.Length.1st Qu. Sepal.Length.Median
## 1      setosa          4.300          4.800          5.000
## 2 versicolor          4.900          5.600          5.900
## 3 virginica          4.900          6.225          6.500
##      Sepal.Length.Mean Sepal.Length.3rd Qu. Sepal.Length.Max.
## 1          5.006          5.200          5.800
## 2          5.936          6.300          7.000
## 3          6.588          6.900          7.900
##      Sepal.Width.Min. Sepal.Width.1st Qu. Sepal.Width.Median Sepal.Width.Mean
## 1          2.300          3.200          3.400          3.428
## 2          2.000          2.525          2.800          2.770
## 3          2.200          2.800          3.000          2.974
##      Sepal.Width.3rd Qu. Sepal.Width.Max.
## 1          3.675          4.400
## 2          3.000          3.400
## 3          3.175          3.800
```

#### 9.2.5.13 range()

La función `range()` devuelve el mínimo y el máximo.

```
range(iris[, 1])
```

```
## [1] 4.3 7.9
```

```
print(vecManip3)
```

```
## [1] 1 5 6 8 NA 45 NA 14
```

```
range(vecManip3)
```

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

```
range(vecManip3, na.rm = TRUE)
```

```
## [1] 1 45
```

#### 9.2.5.14 unique()

La función `unique()` devuelve los valores únicos de un objeto (sin duplicados).

```
unique(iris[, 1])
```

```
## [1] 5.1 4.9 4.7 4.6 5.0 5.4 4.4 4.8 4.3 5.8 5.7 5.2 5.5 4.5 5.3 7.0 6.4
## [18] 6.9 6.5 6.3 6.6 5.9 6.0 6.1 5.6 6.7 6.2 6.8 7.1 7.6 7.3 7.2 7.7 7.4
## [35] 7.9
```

```
print(vecManip3)
```

```
## [1] 1 5 6 8 NA 45 NA 14
```

```
unique(vecManip3)
```

```
## [1] 1 5 6 8 NA 45 14
```

### 9.3 Otras funciones útiles

No podemos abordar todas las funciones útiles, aquí solo abordaremos ciertas funciones. A lo largo de este libro, se usarán nuevas funciones. Cuando se utiliza una nueva función, nuestro reflejo siempre debe ser el mismo: **consultar la documentación** con la función `help()`.

#### 9.3.1 seq\_along()

La función `seq_along()` se usa para crear un vector del tamaño del objeto relleno y tomando como valores los números de 1 a N (N corresponde al número de elementos del objeto). Esta función nos servirá mucho en el capítulo sobre bucles.

```
print(vecManip3)
```

```
## [1] 1 5 6 8 NA 45 NA 14
```

```
seq_along(vecManip3)
```

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

#### 9.3.2 :

La función `:` permite crear una secuencia desde a hacia b por pasos de 1. Ha sido difícil escribir los capítulos anteriores sin usarlo ya que esta función es muy útil. Aquí están algunos ejemplos.

```
5:10
```

```
## [1] 5 6 7 8 9 10
```

```
head(iris[, c(1, 2, 3, 4)])
```

```
## Sepal.Length Sepal.Width Petal.Length Petal.Width
## 1          5.1          3.5          1.4          0.2
## 2          4.9          3.0          1.4          0.2
## 3          4.7          3.2          1.3          0.2
## 4          4.6          3.1          1.5          0.2
## 5          5.0          3.6          1.4          0.2
## 6          5.4          3.9          1.7          0.4
```

```
head(iris[, 1:4]) # ;-)
```

```
## Sepal.Length Sepal.Width Petal.Length Petal.Width
## 1          5.1          3.5          1.4          0.2
## 2          4.9          3.0          1.4          0.2
## 3          4.7          3.2          1.3          0.2
## 4          4.6          3.1          1.5          0.2
## 5          5.0          3.6          1.4          0.2
## 6          5.4          3.9          1.7          0.4
```

```
miVec01 <- c(1, 2, 3, 4)
miVec01 <- 1:4 # ;-)
-10:12
```

```
## [1] -10 -9 -8 -7 -6 -5 -4 -3 -2 -1 0 1 2 3 4 5 6
## [18] 7 8 9 10 11 12
```

```
5:-5
```

```
## [1] 5 4 3 2 1 0 -1 -2 -3 -4 -5
```

```
paste("X", 1:10, sep = "_")
```

```
## [1] "X_1" "X_2" "X_3" "X_4" "X_5" "X_6" "X_7" "X_8" "X_9" "X_10"
```

### 9.3.3 rep()

La función `rep()` permite repetir elementos.

```
miVec12 <- c(1, 1, 1, 1, 1, 1, 1, 1, 1)
miVec12 <- rep(1, times = 9) # ;-)
rep("Hola", times = 3)
```

```
## [1] "Hola" "Hola" "Hola"
```

```
rep(1:3, time = 3)
```

```
## [1] 1 2 3 1 2 3 1 2 3
```

```
rep(1:3, length.out = 10)
```

```
## [1] 1 2 3 1 2 3 1 2 3 1
```

```
rep(1:3, each = 3)
```

```
## [1] 1 1 1 2 2 2 3 3 3
```

### 9.3.4 seq()

La función `seq()` permite crear una secuencia personalizada.

```
seq(from = 0, to = 1, by = 0.2)
```

```
## [1] 0.0 0.2 0.4 0.6 0.8 1.0
```

```
seq(from = 20, to = 10, length.out = 10)
```

```
## [1] 20.00000 18.88889 17.77778 16.66667 15.55556 14.44444 13.33333
## [8] 12.22222 11.11111 10.00000
```

```
letters[seq(from = 1, to = 26, by = 2)]
```

```
## [1] "a" "c" "e" "g" "i" "k" "m" "o" "q" "s" "u" "w" "y"
```

```
rep(seq(from = 1, to = 2, by = 0.5), times = 3)
```

```
## [1] 1.0 1.5 2.0 1.0 1.5 2.0 1.0 1.5 2.0
```

### 9.3.5 getwd()

La función `getwd()` establece la carpeta de trabajo. Esto corresponde a la ubicación relativa desde la cual R se posiciona para identificar los archivos. Este concepto tendrá sentido cuando veamos cómo importar y exportar datos.

```
getwd()
```

```
## [1] "C:/Users/nous/Documents/Francois/TRAVAIL/GitHub/myRbook_SP"
```

### 9.3.6 setwd()

La función `setwd()` se usa para definir un nuevo directorio de trabajo (carpeta de trabajo).

```
oldWd <- getwd()
print(oldWd)
```

```
## [1] "C:/Users/nous/Documents/Francois/TRAVAIL/GitHub/myRbook_SP"
```

```
setwd("../")
getwd()
```

```
## [1] "C:/Users/nous/Documents/Francois/TRAVAIL/GitHub"
```

```
setwd(oldWd)
getwd()
```

```
## [1] "C:/Users/nous/Documents/Francois/TRAVAIL/GitHub/myRbook_SP"
```



### 9.3.7 list.files()

La función `list.files()` se usa para listar todos los archivos en el directorio de trabajo.

```
list.files(pattern = "(html)$") # html
```

```
## [1] "google_analytics_SP.html"
```

```
list.files(pattern = "(pdf)$") # pdf
```

```
## character(0)
```

### 9.3.8 ls()

Al igual que la función `list.files()` hace posible listar todos los archivos presentes en el directorio de trabajo, la función `ls()` permite listar todos los objetos presentes en el entorno de trabajo de R.

```
ls()
```

```
## [1] "aLogic"      "bddInsect"   "characters"  "contrib"
## [5] "dfForMat"    "factor01"    "i"           "irisCopy"
## [9] "j"           "k"           "logicals"    "mdat"
## [13] "miArray"     "miArray02"   "miDf01"      "miDfSub01"
## [17] "miDfSub02"   "miDfSub03"   "miDfSub04"   "miList01"
## [21] "miList02"    "miList0203"  "miList03"    "miList04"
## [25] "miList05"    "miList06"    "miMat"       "miMat01"
## [29] "miMat02"     "miVec01"     "miVec02"     "miVec03"
## [33] "miVec04"     "miVec05"     "miVec06"     "miVec07"
## [37] "miVec08"     "miVec09"     "miVec10"     "miVec11"
## [41] "miVec12"     "miVec13"     "miVec14"     "miVec15"
## [45] "miVec20"     "miVec21"     "miVec22"     "miVec23"
## [49] "miVec24"     "miVec25"     "miVecArr"    "miVecArr02"
## [53] "miVecConf"   "miVecNA"     "miVecOp"     "msg"
## [57] "myCol"       "myRow"       "myText"      "myText2"
## [61] "myText3"     "myText4"     "myText5"     "nbrRep"
## [65] "newVec"      "newVec2"     "numbers"     "oldWd"
## [69] "opAriDf"     "sumIntDou"   "sumIntInt"   "termino01"
## [73] "termino02"   "vecForMat01" "vecForMat02" "vecForMat03"
## [77] "vecForMatrix" "vecManip"    "vecManip2"   "vecManip3"
```

```
zzz <- "a new object"
ls()
```

```
## [1] "aLogic"      "bddInsect"   "characters"  "contrib"
## [5] "dfForMat"    "factor01"    "i"           "irisCopy"
## [9] "j"           "k"           "logicals"    "mdat"
## [13] "miArray"     "miArray02"   "miDf01"      "miDfSub01"
## [17] "miDfSub02"   "miDfSub03"   "miDfSub04"   "miList01"
## [21] "miList02"    "miList0203"  "miList03"    "miList04"
## [25] "miList05"    "miList06"    "miMat"       "miMat01"
## [29] "miMat02"     "miVec01"     "miVec02"     "miVec03"
## [33] "miVec04"     "miVec05"     "miVec06"     "miVec07"
```

```
## [37] "miVec08"      "miVec09"      "miVec10"      "miVec11"
## [41] "miVec12"      "miVec13"      "miVec14"      "miVec15"
## [45] "miVec20"      "miVec21"      "miVec22"      "miVec23"
## [49] "miVec24"      "miVec25"      "miVecArr"     "miVecArr02"
## [53] "miVecConf"    "miVecNA"      "miVecOp"      "msg"
## [57] "myCol"        "myRow"        "myText"       "myText2"
## [61] "myText3"      "myText4"      "myText5"      "nbrRep"
## [65] "newVec"       "newVec2"      "numbers"      "oldWd"
## [69] "opAriDf"      "sumIntDou"    "sumIntInt"    "termino01"
## [73] "termino02"    "vecForMat01"  "vecForMat02"  "vecForMat03"
## [77] "vecForMatrix" "vecManip"     "vecManip2"    "vecManip3"
## [81] "zzz"
```

### 9.3.9 rm()

La función `rm()` permite eliminar un objeto presente en el entorno de trabajo de R.

```
rm(zzz)
ls()
```

```
## [1] "aLogic"      "bddInsect"    "characters"   "contrib"
## [5] "dfForMat"    "factor01"     "i"            "irisCopy"
## [9] "j"           "k"            "logicals"     "mdat"
## [13] "miArray"     "miArray02"    "miDf01"       "miDfSub01"
## [17] "miDfSub02"   "miDfSub03"    "miDfSub04"    "miList01"
## [21] "miList02"    "miList0203"   "miList03"     "miList04"
## [25] "miList05"    "miList06"     "miMat"        "miMat01"
## [29] "miMat02"     "miVec01"      "miVec02"      "miVec03"
## [33] "miVec04"     "miVec05"      "miVec06"      "miVec07"
## [37] "miVec08"     "miVec09"      "miVec10"      "miVec11"
## [41] "miVec12"     "miVec13"      "miVec14"      "miVec15"
## [45] "miVec20"     "miVec21"      "miVec22"      "miVec23"
## [49] "miVec24"     "miVec25"      "miVecArr"     "miVecArr02"
## [53] "miVecConf"   "miVecNA"      "miVecOp"      "msg"
## [57] "myCol"       "myRow"        "myText"       "myText2"
## [61] "myText3"     "myText4"      "myText5"      "nbrRep"
## [65] "newVec"      "newVec2"      "numbers"      "oldWd"
## [69] "opAriDf"     "sumIntDou"    "sumIntInt"    "termino01"
## [73] "termino02"   "vecForMat01"  "vecForMat02"  "vecForMat03"
## [77] "vecForMatrix" "vecManip"     "vecManip2"    "vecManip3"
```

## 9.4 Algunos ejercicios para practicar

Aquí hay algunos ejercicios para mejorar el uso de las funciones y aprender nuevas gracias a la documentación. Algunos ejercicios son difíciles, podremos volver a resolverlos más tarde.

### 9.4.1 Secuencias

#### 9.4.1.1 Vamos a reproducir las siguientes secuencias:

```
-3 -4 -5 -6 -7 -8 -9 -10 -11
```

```
-3 -1 1 3 5 7 9 11
```

```
3.0 3.2 3.4 3.6 3.8 4.0
```

```
20 18 16 14 12 10 8 6
```

```
"a" "f" "k" "p" "u" "z"
```

```
"a" "a" "a" "a" "a" "f" "f" "f" "f" "k" "k" "k" "k" "k" "p" "p" "p" "p" "p" "u" "u" "u" "u" "u" "z" "z" "z" "z" "z"
```

#### 9.4.1.2 Posibles soluciones (porque siempre hay varias soluciones):

```
-3:-11
```

```
## [1] -3 -4 -5 -6 -7 -8 -9 -10 -11
```

```
seq(from = -3, to = 11, by = 2)
```

```
## [1] -3 -1 1 3 5 7 9 11
```

```
seq(from = 3.0, to = 4.0, by = 0.2)
```

```
## [1] 3.0 3.2 3.4 3.6 3.8 4.0
```

```
letters[seq(from = 1, to = 26, by = 5)]
```

```
## [1] "a" "f" "k" "p" "u" "z"
```

```
letters[rep(seq(from = 1, to = 26, by = 5), each = 5)]
```

```
## [1] "a" "a" "a" "a" "a" "f" "f" "f" "f" "f" "k" "k" "k" "k" "k" "p" "p"
```

```
## [18] "p" "p" "p" "u" "u" "u" "u" "u" "u" "z" "z" "z" "z" "z"
```

#### 9.4.2 Estadísticas descriptivas

En el conjunto de datos `iris`, ¿cuántos valores de ancho del sépalo son mayores que 3? Entre 2.8 y 3.2?

¿Cómo se puede visualizar la distribución de datos (función `table()`)?

¿Cuáles son los 10 valores más pequeños?

¿Cómo se calcula el intervalo que contiene el 90% de los valores?

Si la distribución de los datos era Normal, ¿cuál sería el valor teórico de este intervalo del 90% (función `qnorm()`)?

Soluciones:

```
length(iris$Sepal.Width[iris$Sepal.Width > 3])
```

```
## [1] 67
```

```
length(iris$Sepal.Width[iris$Sepal.Width > 2.8 &
  iris$Sepal.Width < 3.2])
```

```
## [1] 47
```

```
table(iris$Sepal.Width)
```

```
##
##  2 2.2 2.3 2.4 2.5 2.6 2.7 2.8 2.9  3 3.1 3.2 3.3 3.4 3.5 3.6 3.7 3.8
##  1  3  4  3  8  5  9 14 10 26 11 13  6 12  6  4  3  6
## 3.9  4 4.1 4.2 4.4
##  2  1  1  1  1
```

```
table(round(iris$Sepal.Width))
```

```
##
##  2  3  4
## 19 106 25
```

```
irisSepWCopy <- iris$Sepal.Width
irisSepWCopy <- irisSepWCopy[order(irisSepWCopy)]
head(irisSepWCopy, n = 10)
```

```
## [1] 2.0 2.2 2.2 2.2 2.3 2.3 2.3 2.3 2.4 2.4
```

```
quantile(irisSepWCopy, probs = c(0.05, 0.95))
```

```
##  5%  95%
## 2.345 3.800
```

```
qnorm(
  p = c(0.05, 0.95),
  mean = mean(irisSepWCopy),
  sd = sd(irisSepWCopy)
)
```

```
## [1] 2.340397 3.774270
```

## 9.5 Escribir una función

Cuando reproducimos las mismas operaciones varias veces, el código se vuelve difícil de escribir y de mantener porque si tenemos que hacer una modificación, tendremos que repetirla cada vez que la usemos. Esto es un signo de la necesidad de usar una **función**. En el siguiente ejemplo, sera largo modificar el código si queremos agregar +45 en lugar de +20 para cada línea.

```
35 + 20
```

```
## [1] 55
```

```
758 + 20
```

```
## [1] 778
```

```
862 + 20
```

```
## [1] 882
```

```
782 + 20
```

```
## [1] 802
```

Como todas las funciones básicas de R, nuestras funciones tendrán un **nombre y argumentos**. Al igual que con los nombres de los objetos y los nombres de los archivos, es importante elegir bien el nombre de nuestra función (ver la sección sobre objetos). Para crear una función utilizaremos la función `function()` que toma como argumento los argumentos de nuestra función. La función devolverá el resultado deseado. Por defecto, el resultado devuelto es el último utilizado, pero es mejor usar la función `return()`. La siguiente función `addX()` toma como argumento `x` y devuelve `x + 20`.

```
addX <- function(x){  
  return(x + 20)  
}
```

Nuestro código se convierte en:

```
addX(35)
```

```
## [1] 55
```

```
addX(758)
```

```
## [1] 778
```

```
addX(862)
```

```
## [1] 882
```

```
addX(782)
```

```
## [1] 802
```

Si queremos cambiar el código para agregar 45 en lugar de 20, simplemente cambiamos la función `addX()`.

```
addX <- function(x){  
  return(x + 45)  
}  
addX(35)
```

```
## [1] 80
```

```
addX(758)
```

```
## [1] 803
```

```
addX(862)
```

```
## [1] 907
```

```
addX(782)
```

```
## [1] 827
```

Aquí podríamos haber usado el formato vector para evitar la repetición, pero eso no siempre es posible.

```
c(35, 758, 862, 782) + 20
```

```
## [1] 55 778 882 802
```

Vamos a escribir una nueva función que contará el número de consonantes y vocales en minúsculas en una palabra. Primero separaremos todas las letras de la palabra con la función `strsplit` (podemos consultar la ayuda para saber más acerca de esta función). Luego contaremos las vocales y las consonantes con la función `length()`. Para la lista de letras, usaremos el objeto `letters` incluido en R que contiene las 26 letras en minúscula (consulte la ayuda con `?letters`).

```
print(letters) # las 26 letras
```

```
## [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q"
## [18] "r" "s" "t" "u" "v" "w" "x" "y" "z"
```

```
countVowelConso <- function(word){ # nombre: countVowelConso ; argumento: word
  wordSplit <- strsplit(word, split = "")[[1]] # separar letras de word
  vowels <- c("a", "e", "i", "o", "u", "y") # las vocales
  numVowel <- length(wordSplit[wordSplit %in% vowels]) # numero de vocales
  consonants <- letters[!letters %in% vowels] # las consonantes
  numConso <- length(wordSplit[wordSplit %in% consonants]) # numero de consonantes
  return(c(numVowel, numConso)) # el resultado de la funcion
}
```

Ahora podemos usar nuestra función.

```
countVowelConso(word = "qwertyuiop azertyuiop")
```

```
## [1] 11 9
```

Esta función se puede modificar mostrando un mensaje más explícito. Aunque en general se debe evitar este tipo de mensaje para evitar sobrecargar las funciones, puede ser útil verificar que todo esté funcionando correctamente (luego lo borraremos).

```
countVowelConso <- function(word){
  wordSplit <- strsplit(word, split = "")[[1]]
  vowels <- c("a", "e", "i", "o", "u", "y")
  numVowel <- length(wordSplit[wordSplit %in% vowels])
  consonants <- letters[!letters %in% vowels]
  numConso <- length(wordSplit[wordSplit %in% consonants])
  print(paste0("Hay ", numVowel, " vocales y ",
    numConso, " consonantes en la palabra '", word, "'."))
  return(c(numVowel, numConso))
}
countVowelConso(word = "qwertyuiop azertyuiop")
```

```
## [1] "Hay 11 vocales y 9 consonantes en la palabra 'qwertyuiop azertyuiop'."
```

```
## [1] 11 9
```

Por otro lado, si usamos `countVowelConso(word = 5)`, se devolverá un error porque nuestra función espera un objeto de tipo `character`. En general, se recomienda *manejar los errores* devueltos por nuestras funciones para que nuestro código sea más fácil de mantener. Aquí simplemente comprobaremos que el argumento sea de tipo `character`, en un vector de tamaño 1. También comentaremos nuestra función para encontrar rápidamente lo que hace (comentario insertado en la primera línea, que a veces encontramos en la última línea de las funciones).

```
countVowelConso <- function(word){ # número de vocales y consonantes
  if(is.vector(word) & is.character(word) & length(word) == 1){
    wordSplit <- strsplit(word, split = "")[[1]]
    vowels <- c("a", "e", "i", "o", "u", "y")
    numVowel <- length(wordSplit[wordSplit %in% vowels])
    consonants <- letters[!letters %in% vowels]
    numConso <- length(wordSplit[wordSplit %in% consonants])
    return(c(numVowel, numConso))
  } else {
    print(paste0("Error: ",
      "argumento 'word' incorrecto (", word, ")"))
  }
}
countVowelConso(word = "qwertyuiop azertyuiop")
```

```
## [1] 11 9
```

```
countVowelConso(word = 5)
```

```
## [1] "Error: argumento 'word' incorrecto (5)"
```

Con R como para cualquier lenguaje de programación, para un problema siempre hay múltiples soluciones. Recordamos la sección sobre tipos de datos (tipo de datos `logical`), así como la sección sobre operadores de comparación que el valor de `TRUE` es 1 y el valor de `FALSE` es 0. Hemos visto anteriormente que la función `% in%` devuelve `TRUE` o `FALSE` para cada elemento del primer objeto dependiendo de su presencia o ausencia en el segundo objeto. Nuestra función podría haber usado otra función en lugar de `length()` para contar vocales y consonantes (función `sum()`).

```
countVowelConsoAlt <- function(word){ # número de vocales y consonantes
  if(is.vector(word) & is.character(word) & length(word) == 1){
    wordSplit <- strsplit(word, split = "")[[1]]
    vowels <- c("a", "e", "i", "o", "u", "y")
    numVowel <- sum(wordSplit %in% vowels) # <- cambio aqui
    consonants <- letters[!letters %in% vowels]
    numConso <- sum(wordSplit %in% consonants) # <- cambio aqui
    return(c(numVowel, numConso))
  } else {
    print(paste0("Error: ",
      "argumento 'word' incorrecto (", word, ")"))
  }
}
countVowelConsoAlt(word = "qwertyuiop azertyuiop")
```

```
## [1] 11 9
```

No existe una solución óptima en absoluto, todo depende de los objetivos deseados. La primera solución puede ser más fácil de entender, y la segunda puede ser más rápida en términos de velocidad de ejecución (repitiendo el uso de la función 10000 veces, el ahorro de tiempo es casi cero en nuestro caso).

```
system.time(replicate(n = 10000, countVowelConso(word = "qwertyuiop azertyuiop")))
```

```
##      user  system elapsed
##    0.15    0.00    0.16
```

```
system.time(replicate(n = 10000, countVowelConsoAlt(word = "qwertyuiop azertyuiop")))
```

```
##      user  system elapsed
##    0.17    0.00    0.18
```

Una función puede tener **valores predeterminados** para sus argumentos. Este es el caso para la mayoría de las funciones existentes. Por defecto, nuestra función ahora contará el número de vocales y consonantes en la palabra `qwerty` (los paréntesis son necesarios incluso en ausencia de argumentos).

```
countVowelConsoAlt <- function(word = "qwerty"){ # número de vocales y consonantes
  if(is.vector(word) & is.character(word) & length(word) == 1){
    wordSplit <- strsplit(word, split = "")[[1]]
    vowels <- c("a", "e", "i", "o", "u", "y")
    numVowel <- sum(wordSplit %in% vowels)
    consonants <- letters[!letters %in% vowels]
    numConso <- sum(wordSplit %in% consonants)
    return(c(numVowel, numConso))
  } else {
    print(paste0("Error: ",
      "argumento 'word' incorrecto (" , word, ")"))
  }
}
countVowelConsoAlt() # no hay que olvidar los paréntesis
```

```
## [1] 2 4
```

R tiene muchas funciones, por lo tanto, antes de comenzar a escribir una nueva función, siempre debemos verificar que ya no exista en la versión básica de R o en los **packages** desarrollado por la comunidad de usuarios. Para esto podemos usar la ayuda con la función `??miBusqueda`, pero también nuestro navegador de Internet.

## 9.6 Otras funciones desarrolladas por la comunidad de usuarios: los *packages*

Un package (o paquete) es un conjunto de archivos que agregaremos a R para usar funciones (o conjuntos de datos) que otras personas hayan desarrollado. Actualmente hay más de 10,000 paquetes en los servidores CRAN de R (CRAN; <https://cran.r-project.org/web/packages/>), más de 1000 en los servidores de BioConductor (para análisis genómicos) y varios cientos en GitHub. Cada paquete hace posible usar nuevas funciones para casi todo ... Por lo tanto, puede ser difícil encontrar el paquete adecuado para lo que queremos lograr, y es importante dedicar tiempo a la búsqueda del paquete adecuado y probar varias soluciones.

Para usar un paquete, primero debemos **instalarlo**, y luego **cargarlo** en nuestra sesión R.

### 9.6.1 Instalar un paquete

Una vez que hemos seleccionado nuestro paquete, podemos descargarlo e instalarlo con la función `install.packages()`, que toma el nombre del paquete entre comillas como argumento (la función tolera la ausencia de comillas, pero es mejor usarlas para que el código sea más legible). Algunos paquetes ya son instalados por defecto con R, como `stats` (que también se carga de forma predeterminada).



```
install.packages("stats") # R statistical functions
```

La instalación de un paquete debe hacerse una vez, luego el paquete está en nuestra computadora.

### 9.6.2 Cargar un paquete

Para poder usar las funciones de un paquete, tenemos que cargarlo en nuestra sesión R. Hay tantos paquetes disponibles que R no cargará todos los que tenemos instalados por defecto, sino solo los que necesitaremos para nuestro estudio actual. Para cargar un paquete usamos la función `library()` o `require()`.

```
library("stats")
```

La carga del paquete debe hacerse cada vez que queremos ejecutar nuestro código, por lo tanto, es una parte integral de nuestro script.

### 9.6.3 Portabilidad del código

Acabamos de ver que la instalación de un paquete solo se debe hacer una vez por computadora, y que la carga de un paquete se debe lograr para cada nueva sesión de R. Si uno cambia de computadora o si compartimos un script con colegas, puede haber errores de ejecución relacionados con la falta de instalación de un paquete. Para superar este problema, se recomienda utilizar una función que verifique si los paquetes necesarios para ejecutar un script están instalados; si es necesario, instálelos y luego cárguelos. Hay muchas funciones para hacer esto en Internet. La solución que proponemos aquí es una mezcla adaptada de diferentes fuentes. No es necesario comprender los detalles de este script por el momento, sino simplemente comprender lo que hace. Este es un ejemplo para el paquete `stats` y `graphics`, dos paquetes que ya están presente con la versión básica de R, pero podemos tratar todos los paquetes disponibles en CRAN; la lista se puede encontrar aquí: [https://cran.r-project.org/web/packages/available\\_packages\\_by\\_name.html](https://cran.r-project.org/web/packages/available_packages_by_name.html).

```
pkgCheck <- function(packages){
  for(x in packages){
    try(if(!require(x, character.only = TRUE)){
      install.packages(x, dependencies = TRUE)
      if(!require(x, character.only = TRUE)){
        stop()
      }
    })
  }
}
pkgCheck(c("stats", "graphics"))
```

Alternativamente, podemos usar la función `.packages()` para listar los paquetes disponibles en el CRAN en orden alfabético.

```
head(.packages(all.available = TRUE), n = 30)
```

```
## [1] "abind"      "ade4"      "ape"      "assertthat" "backports"
## [6] "base64enc"  "BH"        "bindr"    "bindrcpp"   "bitops"
## [11] "bookdown"  "brew"      "broom"    "ca"         "callr"
## [16] "caret"     "cartography" "caTools"  "CircStats"  "classInt"
## [21] "cli"       "clipr"     "clisymbols" "coda"       "colorRamps"
## [26] "colorspace" "commonmark" "cranlogs" "crayon"     "crul"
```

La función `pkgCheck()` asegura la **portabilidad** de nuestros scripts: funcionarán en todas las computadoras sin tener que realizar ningún cambio. Por lo tanto, nuestros scripts pueden adjuntarse, por ejemplo, a nuestros artículos científicos y así garantizar la **reproducibilidad** de nuestros resultados.

## 9.7 Conclusión

Felicitaciones! Ahora sabemos reconocer y usar una función, sabemos cómo buscar ayuda para una función e incluso sabemos escribir nuestras propias funciones. También sabemos que hay muchas funciones desarrolladas por la comunidad de usuarios de R dentro de paquetes (*packages*) que sabemos cómo instalar y cargar, y asegurar la **portabilidad** de nuestros scripts de una computadora a otra (importante para la **reproducibilidad de los resultados**). El próximo capítulo se enfocará en leer y escribir archivos porque nuestros datos suelen estar en archivos de texto u hojas de cálculo.

## Chapter 10

# Importar y exportar datos

### 10.1 Leer datos de un archivo

#### 10.1.1 Transformar datos en formato TXT o CSV

Hay muchas maneras de leer el contenido de un archivo con R. Sin embargo, nos enfocaremos en leer los archivos TXT y CSV que son los más comunes y los más confiables. Con raras excepciones, todos los archivos de datos se pueden transformar fácilmente en formatos TXT y CSV. Esta es la práctica preferida para el análisis de datos con R.

En concreto, desde Microsoft Excel, simplemente vamos a *Archivo*, luego *Guardar como*, seleccionamos el lugar donde queremos guardar nuestro archivo (hablaremos en el siguiente capítulo sobre la gestión de un proyecto R) y luego en el la ventana de copia de seguridad cambiamos el *Tipo* desde XLSX hacia CSV. Desde LibreOffice Calc, simplemente vamos a *Archivo*, luego *Guardar como*, luego seleccionamos el tipo CSV. Es importante saber que el archivo CSV no admite el formato de archivos de hoja de cálculo con, por ejemplo, colores, y que el archivo CSV contiene solo una pestaña. Si tenemos un archivo de hoja de cálculo con varias pestañas, tendremos que guardar tantos archivos CSV como pestañas.

CSV viene del Inglés *Comma-separated values* ([https://es.wikipedia.org/wiki/Valores\\_separados\\_por\\_comas](https://es.wikipedia.org/wiki/Valores_separados_por_comas)) y representa los datos de hoja de cálculo en un formato de texto separado por comas (o punto y coma según el país). Siempre se puede abrir un archivo CSV con software de hoja de cálculo, pero también con un editor de texto simple como el bloc de notas de Windows o con Notepad++. Es preferible abrir archivos CSV con un editor de texto porque las hojas de cálculo tienden a querer cambiar automáticamente los archivos CSV y esto tiene el efecto de dificultar su lectura.

Una vez que se obtiene el archivo TXT o CSV, la lectura del contenido desde R es fácil, aun que requiere un poco de rigor.

#### 10.1.2 Leer un archivo CSV

Esta es la fuente de error más común para los principiantes en R. Es por eso que es importante leer y volver a leer este capítulo y lo siguiente sobre la gestión de un proyecto R con mucha atención.

R funciona en un directorio definido por defecto. Los usuarios de Rstudio u otro entorno de desarrollo especializado para R intentarán usar las opciones disponibles a través de los menús para establecer su directorio de trabajo o cargar el contenido de un archivo. En este libro, estas técnicas nunca se usarán porque no permiten la reproducibilidad de los resultados. Un script debe poder funcionar para todos los sistemas operativos y sin tener en cuenta el entorno de desarrollo del usuario.

El directorio de trabajo por defecto se puede obtener con la función `getwd()` y cambiar con la función `setwd()`.

```
oldWd <- getwd()
print(oldWd)
```

```
## [1] "C:/Users/nous/Documents/Francois/TRAVAIL/GitHub/myRbook_SP"
```

```
setwd("../")
getwd()
```

```
## [1] "C:/Users/nous/Documents/Francois/TRAVAIL/GitHub"
```

```
setwd(oldWd)
getwd()
```

```
## [1] "C:/Users/nous/Documents/Francois/TRAVAIL/GitHub/myRbook_SP"
```

Entonces tenemos cuatro opciones:

- podemos leer el contenido de un archivo indicando a R su ruta completa (limitación para la reproducibilidad de los resultados)
- podemos leer el contenido de un archivo indicando a R su ruta relativa
- podemos mover el archivo en el directorio de trabajo de R
- podemos modificar el directorio de trabajo de R para que coincida con la ubicación de nuestro archivo (con su ruta relativa)

Un ejemplo de una ruta completa sería:

- /home/myName/myFile.csv en un entorno UNIX
- C:/users/myName/myFile.csv bajo un entorno de Windows (tenga cuidado, bajo R utilizamos / y no \ como es el caso por defecto en Windows)

Un camino relativo sería:

- myName/myFiles.csv

Para navegar por las rutas relativas, podemos usar `..` que permite volver al directorio de origen. Por ejemplo, si el directorio de trabajo es `myScripts` y el árbol de mis archivos es:

```
## -myProject
## |-myFiles
## |--data01.csv
## |--data02.csv
## |-myScripts
## |--myFirstScript.R
```

La ruta relativa al archivo `data01.csv` sería `../myFiles/data01.csv`

Entonces, para leer el contenido del archivo `data01.csv`, privilegiaremos la opción 2 (leer el contenido de un archivo que indicando su ruta relativa) o la opción 4 (modificar el directorio de trabajo de R para que coincida con la ubicación de nuestro archivo). En el último caso:

```
myWD <- "../myFiles/"
setwd(myWD)
getwd() # para verificar que estamos en el directorio correcto
list.files() # para verificar que el archivo está aquí
```

El error más común:

```
## Error in setwd("../myFiles/") :
## no se puede cambiar el directorio de trabajo
```

Esto significa que el directorio no existe (se debe verificar que la sintaxis sea correcta y que el directorio exista con esta ruta).

Una vez que el directorio de trabajo está definido correctamente o la ruta relativa al archivo está establecida correctamente, podemos leer el archivo con la función `read.table()`. Algunos usan la función `read.csv()` pero este es solo un caso especial de `read.table()`.

```
myWD <- "../myFiles/"
setwd(myWD)
read.table(file = "data01.csv")
```

o alternativamente:

```
read.table(file = "../myFiles/data01.csv")
```

Si la ruta no se llena correctamente o si el archivo de datos no existe, R devolverá el siguiente error:

```
## Error in file(file, "rt") : incapaz de abrir la conexión
## De más : Warning message:
## In file(file, "rt") :
## incapaz de abrir el archivo '../myFiles/data01.csv' : No such file or directory
```

Si todo está bien, R muestra el contenido del archivo `data01.csv`. Advertencia a los usuarios de Windows porque por defecto no aparece la extensión de los archivos... Así que cuando navegamos a través de directorios con el explorador de archivos, no hay `data01.csv`, pero sólo un archivo `data01`. Es esencial remediar este problema para evitar errores. Para hacer esto, simplemente abrimos las 'Opciones del Explorador de archivos' a través de la tecla 'Windows', luego en la pestaña 'Ver', verificamos que la opción 'Ocultar extensiones de archivos cuyo tipo es conocido' no está marcado.

Consultando la ayuda sobre la función `read.table()`, podemos ver que tiene muchos argumentos. Los principales son:

- `header = FALSE`: ¿el archivo contiene nombres de columna? Si es así, cambiamos el valor a `header = TRUE`
- `sep = ","`: ¿cómo se separan los datos de la tabla? En un archivo CSV es la coma o el punto y coma, así que cambiamos a `sep = ";"` o `sep = ","`
- `dec = "."`: ¿cuál es el separador de los números decimales? Si es la coma, entonces debes cambiar a `dec = ","`

Con estos tres argumentos, la mayoría de los archivos se pueden leer sin ningún problema. En caso de necesidad, la ayuda de esta función es muy completa.

La función `read.table()` devuelve el contenido del archivo como `data.frame`. Para poder usar el contenido del archivo, almacenaremos el `data.frame` en un objeto.

```
myWD <- "../myFiles/"
setwd(myWD)
data01 <- read.table(file = "data01.csv")
str(data01) # verificar el formato de los datos
head(data01) # vverificar los primeros datos
```

El estudio de caso sobre el análisis de datos de datalogger se basa en un archivo CSV. Aquí hay un extracto:

```
bdd <- read.table("myFiles/E05C13.csv", skip = 1, header = TRUE,
  sep = ",", dec = ".", stringsAsFactors = FALSE)
colnames(bdd) <- c("id", "date", "temp")
head(bdd)
```

```
##   id      date    temp
## 1  1 11/12/15 23:00:00 4.973
## 2  2 11/12/15 23:30:00 4.766
## 3  3 11/13/15 00:00:00 4.844
## 4  4 11/13/15 00:30:00 4.844
```

```
## 5 5 11/13/15 01:00:00 5.076
## 6 6 11/13/15 01:30:00 5.282
```

```
tail(bdd)
```

```
##           id           date temp
## 32781 32781 09/25/17 21:00:00 7.091
## 32782 32782 09/25/17 21:30:00 6.914
## 32783 32783 09/25/17 22:00:00 6.813
## 32784 32784 09/25/17 22:30:00 6.611
## 32785 32785 09/25/17 23:00:00 6.331
## 32786 32786 09/25/17 23:30:00 5.385
```

```
str(bdd)
```

```
## 'data.frame': 32786 obs. of 3 variables:
## $ id : int 1 2 3 4 5 6 7 8 9 10 ...
## $ date: chr "11/12/15 23:00:00" "11/12/15 23:30:00" "11/13/15 00:00:00" "11/13/15 00:30:00" ...
## $ temp: num 4.97 4.77 4.84 4.84 5.08 ...
```

### 10.1.3 Leer un archivo de texto

La función más simple para leer un archivo que contiene texto es `readlines()`. Aquí hay un ejemplo con el archivo `README.md` de este libro, que se encuentra en GitHub.

```
readmeGitHub <- "https://raw.githubusercontent.com/frareb/myRBook_SP/master/README.md"
readLines(readmeGitHub)
```

```
## [1] "# myRBook_SP"
## [2] "Aquí se encuentra el código fuente del libro *Aprender R: iniciación y perfeccionamiento*, c
```

También está la función `scan()` que devuelve todas las palabras separadas por espacios. Podemos consultar la ayuda para obtener más información.

```
scan(readmeGitHub, what = "character")
```

```
## [1] "#" "myRBook_SP" "Aquí"
## [4] "se" "encuentra" "el"
## [7] "código" "fuente" "del"
## [10] "libro" "*Aprender" "R:"
## [13] "iniciación" "y" "perfeccionamiento*,"
## [16] "construido" "con" "bookdown."
```

## 10.2 Guardar datos para R

A veces es útil poder guardar un objeto R para poder reutilizarlo más tarde. Este es el caso, por ejemplo, cuando el tiempo de cálculo para llegar a un resultado es muy largo, o cuando queremos liberar espacio en la RAM. Para hacer esto, existe la función `save()` que toma como argumento principal el nombre de los objetos que queremos guardar.

El objeto guardado se almacenará en un archivo. Por convención, es bueno dar como nombre de extensión `.RData` a los archivos que contienen objetos R, preferir un solo objeto por archivo, y dar el nombre del objeto como nombre del archivo.

```
myObject <- 5
ls(pattern = "myObject")
```

```
## [1] "myObject"
```

```
save(myObject, file = "myFiles/myObject.RData")
rm(myObject)
ls(pattern = "myObject")
```

```
## character(0)
```

Si necesitamos el objeto guardado en el archivo, podemos volver a cargarlo con la función `load()`.

```
ls(pattern = "myObject")
```

```
## character(0)
```

```
load("myFiles/myObject.RData")
ls(pattern = "myObject")
```

```
## [1] "myObject"
```

```
print(myObject)
```

```
## [1] 5
```

## 10.3 Exportar datos

La mejor forma de comunicar sus resultados o datos es enviar sus scripts y archivos de datos. A veces esto no es posible o no es adecuado, y puede ser útil exportar sus datos en un archivo de texto o CSV. Para hacer esto, existe la función genérica `write()` y la función `write.table()` para `data.frame`.

Por ejemplo, crearemos un `data.frame` con los números del 1 al 26 y las letras correspondientes, luego los guardaremos en un archivo CSV, luego volveremos a leer los datos contenidos en este archivo.

```
dfLetters <- data.frame(num = 1:26, letters = letters)
write.table(dfLetters, file = "myFiles/dfLetters.csv",
  sep = ",", col.names = TRUE, row.names = FALSE)
read.table(file = "myFiles/dfLetters.csv", header = TRUE, sep = ",")
```

```
##      num letters
## 1      1      a
## 2      2      b
## 3      3      c
## 4      4      d
## 5      5      e
## 6      6      f
## 7      7      g
## 8      8      h
## 9      9      i
## 10    10      j
```

```
## 11 11      k
## 12 12      l
## 13 13      m
## 14 14      n
## 15 15      o
## 16 16      p
## 17 17      q
## 18 18      r
## 19 19      s
## 20 20      t
## 21 21      u
## 22 22      v
## 23 23      w
## 24 24      x
## 25 25      y
## 26 26      z
```

## 10.4 Conclusión

Felicitaciones! Ahora sabemos cómo leer datos de un archivo de texto o CSV, guardar y cargar datos de RData, y escribir en un archivo. El error más común entre los principiantes en R es la lectura de archivos de datos. Es por eso que este capítulo es para leer y volver a leer sin moderación.



## Chapter 11

# Algorítmico

### 11.1 Pruebas lógicas con if

Si queremos realizar una operación diferente según una condición, podemos configurar una prueba lógica del tipo **SI esto ENTONCES esto SINO esto**. Con R esto dará como resultado la función `if(cond) cons.express alt.expr` como se muestra en la función `help`.

```
myVar <- 2
if(myVar < 3) print("myVar < 3")
```

```
## [1] "myVar < 3"
```

```
if(myVar < 3) print("myVar < 3") else print("myVar > 3")
```

```
## [1] "myVar < 3"
```

Cuando hay varias líneas de código para ejecutar basadas en la prueba lógica, o simplemente para hacer que el código sea más fácil de leer, utilizamos varias líneas con `{}` y con indentación.

```
myVar <- 2
myResult <- 0
if(myVar < 3){
  print("myVar < 3")
  myResult <- myVar + 10
} else {
  print("myVar > 3")
  myResult <- myVar - 10
}
```

```
## [1] "myVar < 3"
```

```
print(myResult)
```

```
## [1] 12
```

En este ejemplo definimos una variable `myVar`. Si esta variable es menor que 3, la variable `myResult` se establece en `myVar + 10`, y de lo contrario `myResult` se establece en `myVar - 10`.

Ya hemos visto el uso de la prueba lógica `if` en el capítulo sobre las funciones. Hemos probado si la variable ingresada como argumento en nuestra función era de tipo `character`.

```
myVar <- "qwerty"
if(is.character(myVar)){
  print("ok")
} else {
  print("error")
}
```

```
## [1] "ok"
```

También podemos anidar pruebas lógicas entre sí.

```
myVar <- TRUE
if(is.character(myVar)){
  print("myVar: character")
} else {
  if(is.numeric(myVar)){
    print("myVar: numeric")
  } else {
    if(is.logical(myVar)){
      print("myVar: logical")
    } else {
      print("myVar: ...")
    }
  }
}
```

```
## [1] "myVar: logical"
```

También es posible estipular varias condiciones, como vimos en el capítulo sobre operadores de comparación.

```
myVar <- 2
if(myVar > 1 & myVar < 50){
  print("ok")
}
```

```
## [1] "ok"
```

En este ejemplo, `myVar` está en formato `numeric`, por lo que la primera condición (`> 1`) y la segunda condición (`< 50`) son verificables. Por otro lado, si asignamos una variable de tipo `character` a `myVar` entonces R transformará 0 y 10 en objetos de tipo `character` y probará si `myVar > "1"` y después si `myVar < "50"` basándose en la clasificación alfabética. En el siguiente ejemplo, "azerty" no está ubicado según el orden alfabético entre "1" y "50", pero para "2azerty" es el caso, lo que resulta problemático.

```
myVar <- "azerty"
limInit <- 1
limEnd <- 50
if(myVar > limInit & myVar < limEnd){
  print(paste0(myVar, " is between ", limInit, " and ", limEnd, "."))
} else {
  print(paste0(myVar, " not between ", limInit, " and ", limEnd, "."))
}
```

```
## [1] "azerty not between 1 and 50."
```

```
myVar <- "2azerty"
if(myVar > limInit & myVar < limEnd){
  print(paste0(myVar, " is between ", limInit, " and ", limEnd, "."))
} else {
  print(paste0(myVar, " not between ", limInit, " and ", limEnd, "."))
}
```

```
## [1] "2azerty is between 1 and 50."
```

Entonces, lo que nos gustaría hacer es probar si `myVar` está en formato `numeric`, y entonces solo si es el caso probar las siguientes condiciones.

```
myVar <- "2azerty"
if(is.numeric(myVar)){
  if(myVar > limInit & myVar < limEnd){
    print(paste0(myVar, " is between ", limInit, " and ", limEnd, "."))
  } else {
    print(paste0(myVar, " not between ", limInit, " and ", limEnd, "."))
  }
} else {
  print(paste0("Object ", myVar, " is not numeric"))
}
```

```
## [1] "Object 2azerty is not numeric"
```

A veces es posible que necesitemos probar una primera condición y luego una segunda condición solo si la primera es verdadera en la misma prueba. Por ejemplo, para un sitio nos gustaría saber si hay una sola especie y probar si su abundancia es mayor que 10. Imagine un conjunto de datos con abundancia de vectores. Probaremos el número de especies con la función `length()`.

```
mySpecies <- c(15, 14, 20, 12)
if(length(mySpecies) == 1 & mySpecies > 10){
  print("ok!")
}
## Warning message:
## In if (length(mySpecies) == 1 & mySpecies > 10) { :
## the condition has length > 1 and only the first element will be used
```

R devuelve un error porque no puede dentro de una prueba lógica con `if()` verificar la segunda condición. De hecho, `mySpecies > 10` devuelve `TRUE TRUE TRUE TRUE TRUE`. Podemos separar el código en dos condiciones:

```
mySpecies <- c(15, 14, 20, 12)
if(length(mySpecies) == 1){
  if(mySpecies > 10){
    print("ok!")
  }
}
```

Una alternativa más elegante es decirle a R que verifique la segunda condición solo si la primera es verdadera. Para eso podemos usar `&&` en lugar de `&`.

```
mySpecies <- c(15, 14, 20, 12)
if(length(mySpecies) == 1 && mySpecies > 10){
  print("ok!")
}
mySpecies <- 15
if(length(mySpecies) == 1 && mySpecies > 10){
  print("ok!")
}
```

```
## [1] "ok!"
```

```
mySpecies <- 5
if(length(mySpecies) == 1 && mySpecies > 10){
  print("ok!")
}
```

Con & R comprobará todas las condiciones, y con && R tomará cada condición una después de la otra y continuará solo si es verdadera. Esto puede parecer anecdótico, pero es bueno saber la diferencia entre & y && porque a menudo los encontramos en los códigos disponibles en Internet o en los paquetes.

## 11.2 Pruebas lógicas con switch

La función `switch()` es una variante de `if()` que es útil cuando tenemos muchas opciones posibles para la misma expresión. El siguiente ejemplo muestra cómo transformar el código usando `if()` a `switch()`.

```
x <- "aa"
if(x == "a"){
  result <- 1
}
if(x == "aa"){
  result <- 2
}
if(x == "aaa"){
  result <- 3
}
if(x == "aaaa"){
  result <- 4
}
print(result)
```

```
## [1] 2
```

```
x <- "aa"
switch(x,
  a = result <- 1,
  aa = result <- 2,
  aaa = result <- 3,
  aaaa = result <- 4)
print(result)
```

```
## [1] 2
```

## 11.3 El bucle for

En programación, cuando tenemos que repetir la misma línea de código varias veces, es un signo que indica que debemos usar un **bucle**. Un bucle es una forma de iterar sobre un conjunto de objetos (o los elementos de un objeto) y repetir una operación. Imaginamos un `data.frame` con mediciones de datos de campo en dos fechas.

```
bdd <- data.frame(date01 = rnorm(n = 100, mean = 10, sd = 1),
                  date02 = rnorm(n = 100, mean = 10, sd = 1))
print(head(bdd))
```

```
##      date01    date02
## 1 11.174321  9.715279
## 2  9.812169 11.789370
## 3  9.963385  8.627320
## 4  9.217849 10.404683
## 5  8.983468 12.236569
## 6  8.670386 11.051223
```

Nos gustaría cuantificar la diferencia entre la primera y la segunda fecha, luego poner un indicador para saber si esta diferencia es pequeña o grande, por ejemplo, con un umbral arbitrario de 3. Entonces, para cada línea podríamos hacer:

```
bdd$dif <- NA
bdd$isDifBig <- NA

bdd$dif[1] <- sqrt((bdd$date01[1] - bdd$date02[1])^2)
bdd$dif[2] <- sqrt((bdd$date01[2] - bdd$date02[2])^2)
bdd$dif[3] <- sqrt((bdd$date01[3] - bdd$date02[3])^2)
# ...
bdd$dif[100] <- sqrt((bdd$date01[100] - bdd$date02[100])^2)

if(bdd$dif[1] > 3){
  bdd$isDifBig[1] <- "big"
}else{
  bdd$isDifBig[1] <- "small"
}
if(bdd$dif[2] > 3){
  bdd$isDifBig[2] <- "big"
}else{
  bdd$isDifBig[2] <- "small"
}
if(bdd$dif[3] > 3){
  bdd$isDifBig[3] <- "big"
}else{
  bdd$isDifBig[3] <- "small"
}
# ...
if(bdd$dif[100] > 3){
  bdd$isDifBig[100] <- "big"
}else{
  bdd$isDifBig[100] <- "small"
}
```

Esta forma de hacer las cosas sería extremadamente tediosa de lograr, y casi imposible de lograr si la tabla contuviera 1000 o 100000 líneas. Puede parecer lógico querer iterar sobre las líneas de nuestro `data.frame` para obtener las nuevas columnas.

Es lo que vamos a hacer aun que no es la solución que retendremos más adelante.

Vamos a usar un bucle `for()`. El bucle `for()` recorrerá los elementos de un objeto que vamos a dar como argumento. Por ejemplo, aquí hay un bucle que para todos los números del 3 al 9 calculará su valor al cuadrado. El valor actual del número está simbolizado por un objeto que puede tomar el nombre que queramos (aquí será `i`).

```
for(i in c(3, 4, 5, 6, 7, 8, 9)){
  print(i^2)
}
```

```
## [1] 9
## [1] 16
## [1] 25
## [1] 36
## [1] 49
## [1] 64
## [1] 81
```

Eso podemos mejorar usando la función `:`.

```
for(i in 3:9){
  print(i^2)
}
```

El bucle `for()` puede iterar sobre todos los tipos de elementos.

```
nChar <- c("a", "z", "e", "r", "t", "y")
for(i in nChar){
  print(i)
}
```

```
## [1] "a"
## [1] "z"
## [1] "e"
## [1] "r"
## [1] "t"
## [1] "y"
```

Volvamos a nuestro caso. Vamos a iterar sobre el número de líneas de nuestro `data.frame` `bdd`. Antes de eso crearemos las columnas `dif` y `isDifBig` con los valores NA. Luego usaremos la función `nrow()` para encontrar el número de líneas.

```
bdd$dif <- NA
bdd$isDifBig <- NA
for(i in 1:nrow(bdd)){
  bdd$dif[i] <- sqrt((bdd$date01[i] - bdd$date02[i])^2)
  if(bdd$dif[i] > 3){
    bdd$isDifBig[i] <- "big"
  }else{
    bdd$isDifBig[i] <- "small"
  }
}
print(head(bdd, n = 20))
```

```
##      date01    date02      dif isDifBig
## 1  11.174321  9.715279 1.45904152    small
```

```
## 2  9.812169 11.789370 1.97720105    small
## 3  9.963385  8.627320 1.33606446    small
## 4  9.217849 10.404683 1.18683366    small
## 5  8.983468 12.236569 3.25310075      big
## 6  8.670386 11.051223 2.38083711    small
## 7 11.961661 11.143027 0.81863416    small
## 8 10.905488 11.285875 0.38038785    small
## 9  9.144415  9.469015 0.32460018    small
##10 11.055156 10.341123 0.71403335    small
##11 10.310937  9.826348 0.48458892    small
##12 10.414327  9.895231 0.51909643    small
##13 10.976084 11.035409 0.05932488    small
##14 10.828736  9.705851 1.12288503    small
##15 10.514551 11.106574 0.59202244    small
##16 11.642322  9.704503 1.93781876    small
##17 10.543551  9.769919 0.77363173    small
##18 12.103575  9.726243 2.37733140    small
##19  8.052176 11.569782 3.51760572      big
##20  8.770527  9.455768 0.68524068    small
```

En la práctica, esta no es la mejor manera de realizar este ejercicio porque se trata de cálculos simples en vectores contenidos en un `data.frame`. R es particularmente potente para realizar operaciones en vectores. Donde sea posible, siempre tenemos que enfocarnos en operaciones vectoriales. Aquí nuestro código se convierte en:

```
bdd$dif <- sqrt((bdd$date01 - bdd$date02)^2)
bdd$isDifBig <- "small"
bdd$isDifBig[bdd$dif > 3] <- "big"
print(head(bdd, n = 20))
```

```
##      date01      date02      dif isDifBig
## 1 11.174321  9.715279 1.45904152    small
## 2  9.812169 11.789370 1.97720105    small
## 3  9.963385  8.627320 1.33606446    small
## 4  9.217849 10.404683 1.18683366    small
## 5  8.983468 12.236569 3.25310075      big
## 6  8.670386 11.051223 2.38083711    small
## 7 11.961661 11.143027 0.81863416    small
## 8 10.905488 11.285875 0.38038785    small
## 9  9.144415  9.469015 0.32460018    small
##10 11.055156 10.341123 0.71403335    small
##11 10.310937  9.826348 0.48458892    small
##12 10.414327  9.895231 0.51909643    small
##13 10.976084 11.035409 0.05932488    small
##14 10.828736  9.705851 1.12288503    small
##15 10.514551 11.106574 0.59202244    small
##16 11.642322  9.704503 1.93781876    small
##17 10.543551  9.769919 0.77363173    small
##18 12.103575  9.726243 2.37733140    small
##19  8.052176 11.569782 3.51760572      big
##20  8.770527  9.455768 0.68524068    small
```

La mayoría de los ejemplos que se pueden encontrar en Internet sobre el bucle `for()` pueden reemplazarse por operaciones vectoriales. Aquí hay algunos ejemplos adaptados de varias fuentes:

```
# prueba si los números son pares
# [1] FOR
x <- sample(1:100, size = 20)
count <- 0
for (val in x) {
  if(val %% 2 == 0){
    count <- count + 1
  }
}
print(count)
```

```
## [1] 8
```

```
# [2] VECTOR
sum(x %% 2 == 0)
```

```
## [1] 8
```

```
# calcular cuadrados
# [1] FOR
x <- rep(0, 20)
for (j in 1:20){
  x[j] <- j^2
}
print(x)
```

```
## [1] 1 4 9 16 25 36 49 64 81 100 121 144 169 196 225 256 289
## [18] 324 361 400
```

```
# [2] VECTOR
(1:20)^2
```

```
## [1] 1 4 9 16 25 36 49 64 81 100 121 144 169 196 225 256 289
## [18] 324 361 400
```

```
# repetir una tirada de dados y promediar
# [1] FOR
ntrials = 1000
trials = rep(0, ntrials)
for (j in 1:ntrials){
  trials[j] = sample(1:6, size = 1)
}
mean(trials)
```

```
## [1] 3.458
```

```
# [2] VECTOR
mean(sample(1:6, ntrials, replace = TRUE))
```

```
## [1] 3.548
```

Es un buen ejercicio explorar los muchos ejemplos disponibles en Internet en el bucle `for()` e intentar convertirlos en operaciones vectoriales. Esto nos permite adquirir buenos reflejos de programación con R. El bucle `for()` es muy útil, por ejemplo,



para leer varios archivos y tratar la información que contienen de la misma manera, hacer gráficos, o Cuando las operaciones vectoriales se vuelven tediosas. Imagina una matriz de 10 columnas y 100 líneas. Queremos la suma de cada línea (veremos cómo hacer con la función `apply()` mas adelante).

```
myMat <- matrix(sample(1:100, size = 1000, replace = TRUE), ncol = 10)
# VECTOR
sumRow <- myMat[, 1] + myMat[, 2] + myMat[, 3] + myMat[, 4] +
  myMat[, 5] + myMat[, 6] + myMat[, 7] + myMat[, 8] +
  myMat[, 9] + myMat[, 10]
print(sumRow)
```

```
## [1] 290 501 576 541 468 511 549 590 550 435 531 575 603 498 420 406 547
## [18] 418 385 575 469 547 468 415 521 530 484 617 377 718 477 590 426 652
## [35] 511 510 283 443 511 652 507 475 603 529 558 396 622 615 447 465 462
## [52] 569 456 485 536 549 632 510 512 587 538 667 569 544 669 376 387 419
## [69] 625 472 590 461 506 388 530 267 419 632 573 462 360 364 414 559 536
## [86] 587 578 517 696 719 579 476 495 542 348 613 508 694 474 511
```

```
# FOR
sumRow <- rep(NA, times = nrow(myMat))
for(j in 1:nrow(myMat)){
  sumRow[j] <- sum(myMat[j, ])
}
print(sumRow)
```

```
## [1] 290 501 576 541 468 511 549 590 550 435 531 575 603 498 420 406 547
## [18] 418 385 575 469 547 468 415 521 530 484 617 377 718 477 590 426 652
## [35] 511 510 283 443 511 652 507 475 603 529 558 396 622 615 447 465 462
## [52] 569 456 485 536 549 632 510 512 587 538 667 569 544 669 376 387 419
## [69] 625 472 590 461 506 388 530 267 419 632 573 462 360 364 414 559 536
## [86] 587 578 517 696 719 579 476 495 542 348 613 508 694 474 511
```

En conclusión, se recomienda no usar el bucle `for()` con R siempre que sea posible, y en este capítulo veremos alternativas como los bucles familiares `apply()`.

## 11.4 El bucle while

El bucle `while()`, a diferencia del bucle `for()`, significa *MIENTRAS*. Mientras no se cumpla una condición, el bucle continuará ejecutándose. Atención porque en caso de error, podemos programar fácilmente bucles que nunca terminan. Este bucle es menos común que el bucle `for()`. Tomemos un ejemplo:

```
i <- 0
while(i < 10){
  print(i)
  i <- i + 1
}
```

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

```
## [1] 5
## [1] 6
## [1] 7
## [1] 8
## [1] 9
```

En este ejemplo, la variable `i` tiene como valor inicial 0. MIENTRAS QUE `i < 10`, mostramos `i` con `print()`. Para que este bucle finalice, no olvidamos cambiar el valor de `i`, esto se hace con la línea `i <- i + 1`. Cuando la condición `i < 10` ya no se cumple, el bucle se detiene.

El bucle `while()` es muy útil para crear scripts que realizarán cálculos en variables cuyo valor cambia con el tiempo. Por ejemplo, imaginamos un número entre 0 y 10000 y un generador aleatorio que intentará determinar el valor de este número. Si queremos limitar los intentos de R a 2 segundos, podemos escribir el siguiente script (que debería funcionar cada vez en una computadora de escritorio típica que pueda realizar fácilmente 35000 pruebas en 2 segundos):

```
myNumber <- sample(x = 10000, size = 1)
myGuess <- sample(x = 10000, size = 1)
startTime <- Sys.time()
numberGuess <- 0
while(Sys.time() - startTime < 2){
  if(myGuess == myNumber){
    numberGuess <- numberGuess + 1
    print("Number found !")
    print(paste0("And I have plenty of time left: ",
      round(2 - as.numeric(Sys.time() - startTime), digits = 2),
      " sec"))
    break
  }else{
    myGuess <- sample(x = 10000, size = 1)
    numberGuess <- numberGuess + 1
  }
}
```

```
## [1] "Number found !"
## [1] "And I have plenty of time left: 1.79 sec"
```

En este script generamos un número aleatorio para adivinar con la función `sample()`, y cada uno de los intentos con la misma función `sample()`. Luego usamos la función `Sys.time()` (con una `S` mayúscula a `Sys`), para saber la hora de inicio del bucle. Siempre que la diferencia entre cada iteración del bucle y la hora de inicio sea inferior a 2 segundos, el bucle `while()` verificará si el número correcto estaba adivinando en la prueba lógica con `if()` y luego si es el caso nos informa que se encontró el número, y nos indica el tiempo restante antes de los dos segundos. Luego para finalizar el bucle usamos la palabra clave “break” en la que volveremos. En resumen, `break`, permite salir de un bucle. Si no se ha adivinado el número, el bucle realiza otra prueba con la función `sample()`.

Más concretamente, podríamos imaginar algoritmos para explorar un espacio de soluciones a un problema con un tiempo limitado para lograrlo. El bucle `while()` también puede ser útil para que un script se ejecute solo cuando un archivo de otro programa esté disponible ... En la práctica, el bucle `while()` se usa poco con R.

## 11.5 El bucle repeat

El bucle `repeat()` permite repetir una operación sin condiciones para verificar. Para salir de este bucle debemos usar la palabra clave `break`.

```
i <- 1
repeat{
  print(i^2)
  i <- i + 1
  if(i == 5){
    break
  }
}
```

```
## [1] 1
## [1] 4
## [1] 9
## [1] 16
```

Si volvemos al ejemplo anterior, podemos usar un bucle `repeat()` para repetirlo cinco veces.

```
numTry <- 0
repeat{
  myNumber <- sample(x = 10000, size = 1)
  myGuess <- sample(x = 10000, size = 1)
  startTime <- Sys.time()
  numberGuess <- 0
  while(Sys.time() - startTime < 2){
    if(myGuess == myNumber){
      numberGuess <- numberGuess + 1
      print(round(as.numeric(Sys.time() - startTime), digits = 3))
      break
    }else{
      myGuess <- sample(x = 10000, size = 1)
      numberGuess <- numberGuess + 1
    }
  }
  numTry <- numTry + 1
  if(numTry == 5){break}
}
```

```
## [1] 0.863
## [1] 0.387
## [1] 0.031
## [1] 1.372
```

Al igual que el bucle `while()`, el bucle `repeat()` no se usa mucho con R.

## 11.6 next y break

Ya hemos visto la palabra clave `break` que permite salir del bucle actual. Por ejemplo, si buscamos el primer dígito después de 111 que es divisible por 32:

```
myVars <- 111:1000
for(myVar in myVars){
  if(myVar %% 32 == 0){
    print(myVar)
  }
}
```

```

    break
  }
}

```

```
## [1] 128
```

Aunque hemos visto que en la práctica podemos evitar el bucle `for()` con una operación vectorial:

```
(111:1000)[111:1000 %% 32 == 0][1]
```

```
## [1] 128
```

La palabra clave `next` permite pasar a la siguiente iteración de un bucle si se cumple una determinada condición. Por ejemplo, si queremos imprimir las letras del alfabeto sin las vocales:

```

for(myLetter in letters){
  if(myLetter %in% c("a", "e", "i", "o", "u", "y")){
    next
  }
  print(myLetter)
}

```

```

## [1] "b"
## [1] "c"
## [1] "d"
## [1] "f"
## [1] "g"
## [1] "h"
## [1] "j"
## [1] "k"
## [1] "l"
## [1] "m"
## [1] "n"
## [1] "p"
## [1] "q"
## [1] "r"
## [1] "s"
## [1] "t"
## [1] "v"
## [1] "w"
## [1] "x"
## [1] "z"

```

De nuevo podemos evitar el bucle `for()` con:

```
letters[! letters %in% c("a", "e", "i", "o", "u", "y")]
```

```

## [1] "b" "c" "d" "f" "g" "h" "j" "k" "l" "m" "n" "p" "q" "r" "s" "t" "v"
## [18] "w" "x" "z"

```

En conclusión, si usamos bucles, las palabras clave `next` y `break` suelen ser muy útiles, pero siempre que sea posible es mejor usar operaciones vectoriales. Cuando no es posible trabajar con vectores, es mejor usar los bucles del tipo `apply` que son el tema de la siguiente sección.

## 11.7 Los bucles de la familia apply

### 11.7.1 apply

La función `apply()` permite aplicar una función a todos los elementos de un array o un matrix. Por ejemplo, si queremos saber la suma de cada fila de una matriz de 10 columnas y 100 líneas:

```
myMat <- matrix(sample(1:100, size = 1000, replace = TRUE), ncol = 10)
apply(X = myMat, MARGIN = 1, FUN = sum)
```

```
## [1] 473 316 395 464 325 409 584 667 425 525 443 657 366 485 611 576 504
## [18] 468 551 736 580 558 532 589 548 477 497 538 470 533 585 442 401 433
## [35] 515 521 495 455 496 571 474 538 452 695 517 583 539 434 613 504 482
## [52] 559 316 550 508 695 384 511 465 498 595 490 460 565 389 569 328 726
## [69] 374 425 500 328 548 555 509 668 495 608 572 555 418 406 623 633 483
## [86] 489 469 589 593 490 421 419 468 441 424 546 364 691 481 419
```

Si queremos saber la mediana de cada columna, la expresión se convierte en:

```
apply(X = myMat, MARGIN = 2, FUN = median)
```

```
## [1] 52.0 54.0 52.5 54.5 52.0 48.0 51.0 44.5 49.5 53.0
```

El argumento `X` es el objeto en el que el bucle `apply` se repetirá. El argumento `MARGIN` corresponde a la dimensión a tener en cuenta (1 para las filas y 2 para las columnas). El argumento `FUN` es la función a aplicar. En un objeto array, el argumento `MARGIN` puede tomar tantos valores como dimensiones. En este ejemplo, `MARGIN = 1` es el promedio de cada fila - dimensión 1 - (todas las dimensiones combinadas), `MARGIN = 2` es el promedio de cada columna - dimensión 2 - (todas las dimensiones combinadas), y `MARGIN = 3` es el promedio de cada dimensión 3. Debajo cada cálculo se realiza de dos maneras diferentes para explicar su operación.

```
myArr <- array(sample(1:100, size = 1000, replace = TRUE), dim = c(10, 20, 5))
apply(X = myArr, MARGIN = 1, FUN = mean)
```

```
## [1] 51.79 46.35 48.88 51.21 52.71 49.12 49.42 45.41 53.33 53.46
```

```
(apply(myArr[,1], 1, mean) + apply(myArr[,2], 1, mean) +
  apply(myArr[,3], 1, mean) + apply(myArr[,4], 1, mean) +
  apply(myArr[,5], 1, mean))/5
```

```
## [1] 51.79 46.35 48.88 51.21 52.71 49.12 49.42 45.41 53.33 53.46
```

```
apply(X = myArr, MARGIN = 2, FUN = mean)
```

```
## [1] 54.80 52.48 47.58 51.50 46.10 51.08 48.70 45.56 47.42 43.48 45.52
## [12] 52.28 60.68 46.26 52.78 48.58 61.04 46.32 45.62 55.58
```

```
(apply(myArr[,1], 2, mean) + apply(myArr[,2], 2, mean) +
  apply(myArr[,3], 2, mean) + apply(myArr[,4], 2, mean) +
  apply(myArr[,5], 2, mean))/5
```

```
## [1] 54.80 52.48 47.58 51.50 46.10 51.08 48.70 45.56 47.42 43.48 45.52
## [12] 52.28 60.68 46.26 52.78 48.58 61.04 46.32 45.62 55.58
```

```
apply(X = myArr, MARGIN = 3, FUN = mean)
```

```
## [1] 51.625 46.575 49.985 51.200 51.455
```

```
c(mean(myArr[,1]), mean(myArr[,2]), mean(myArr[,3]),
  mean(myArr[,4]), mean(myArr[,5]))
```

```
## [1] 51.625 46.575 49.985 51.200 51.455
```

También podemos calcular el promedio de cada fila y valor de columna (la función luego itera en la dimensión 3):

```
apply(X = myArr, MARGIN = c(1, 2), FUN = mean)
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10] [,11] [,12] [,13]
## [1,] 42.6 49.0 45.0 58.0 67.6 22.2 24.0 51.2 48.8 41.8 62.2 64.8 78.6
## [2,] 27.4 44.6 23.6 43.0 39.0 26.2 38.0 69.4 47.6 41.2 38.6 52.8 77.2
## [3,] 59.8 64.8 61.2 43.8 50.2 59.4 29.6 33.4 41.6 57.4 30.4 42.8 61.0
## [4,] 64.4 62.2 29.2 47.4 45.4 67.0 56.2 52.8 38.8 27.4 65.8 61.2 61.0
## [5,] 58.4 47.2 44.6 44.6 44.8 80.0 60.2 32.6 66.0 64.8 40.4 40.4 65.2
## [6,] 78.0 50.2 33.0 34.2 62.8 34.2 50.2 47.6 52.0 21.6 56.4 28.8 45.4
## [7,] 77.2 56.6 59.4 52.2 46.4 47.4 64.4 24.2 37.0 44.8 49.6 36.6 31.0
## [8,] 34.2 47.8 74.0 61.4 18.8 64.8 35.0 41.0 34.8 35.4 31.0 46.6 55.2
## [9,] 51.6 58.6 32.4 68.6 40.6 53.2 76.0 50.8 48.4 52.2 35.4 72.4 70.8
## [10,] 54.4 43.8 73.4 61.8 45.4 56.4 53.4 52.6 59.2 48.2 45.4 76.4 61.4
##      [,14] [,15] [,16] [,17] [,18] [,19] [,20]
## [1,] 46.0 79.4 31.2 48.4 47.4 49.0 78.6
## [2,] 61.0 51.0 39.0 67.4 44.8 55.2 40.0
## [3,] 36.8 25.6 64.8 63.8 70.6 23.8 56.8
## [4,] 48.2 61.6 52.4 48.8 33.2 41.6 59.6
## [5,] 47.8 61.2 59.0 65.6 35.6 32.0 63.8
## [6,] 34.0 65.4 55.8 83.4 56.2 31.2 62.0
## [7,] 38.6 72.6 37.6 70.2 29.6 58.8 54.2
## [8,] 54.0 34.2 47.0 48.8 51.8 46.4 46.0
## [9,] 46.2 17.4 58.6 55.6 61.6 63.0 53.2
## [10,] 50.0 59.4 40.4 58.4 32.4 55.2 41.6
```

### 11.7.2 lapply

Como se indica en la documentación, `lapply()` devuelve una lista de la misma longitud que `X`, y cada elemento resulta de la aplicación `FUN` al elemento `X` correspondiente. Si `X` es una `list` que contiene `vector` y estamos tratando de obtener el promedio de cada elemento de `list`, podemos usar la función `lapply()`:

```
myList <- list(
  a = sample(1:100, size = 10),
  b = sample(1:100, size = 10),
  c = sample(1:100, size = 10),
  d = sample(1:100, size = 10),
  e = sample(1:100, size = 10)
)
print(myList)
```

```
## $a
## [1] 12 16 81 59 63 74 55 41 1 34
##
## $b
## [1] 43 44 72 69 67 19 52 41 56 13
##
## $c
## [1] 64 90 5 39 24 30 61 20 99 14
##
## $d
## [1] 95 83 6 37 84 24 2 5 57 39
##
## $e
## [1] 50 45 69 75 81 16 59 53 37 56
```

```
lapply(myList, FUN = mean)
```

```
## $a
## [1] 43.6
##
## $b
## [1] 47.6
##
## $c
## [1] 44.6
##
## $d
## [1] 43.2
##
## $e
## [1] 54.1
```

Al igual que con la función `apply()`, podemos pasar argumentos adicionales a la función `lapply()` agregándolos después de la función. Esto es útil, por ejemplo, si nuestra `list` contiene estos valores faltantes `NA` y queremos ignorarlos para calcular los promedios (con el argumento `na.rm = TRUE`).

```
myList <- list(
  a = sample(c(1:5, NA), size = 10, replace = TRUE),
  b = sample(c(1:5, NA), size = 10, replace = TRUE),
  c = sample(c(1:5, NA), size = 10, replace = TRUE),
  d = sample(c(1:5, NA), size = 10, replace = TRUE),
  e = sample(c(1:5, NA), size = 10, replace = TRUE)
)
print(myList)
```

```
## $a
## [1] 2 4 5 5 5 3 5 4 5 4
##
## $b
## [1] 3 5 NA 4 4 2 5 5 2 5
##
## $c
## [1] 5 2 5 3 3 2 2 5 5 5
##
```

```
## $d
## [1] 3 NA 2 2 5 5 1 4 3 NA
##
## $e
## [1] 4 2 2 5 2 NA 2 4 2 1
```

```
lapply(myList, FUN = mean)
```

```
## $a
## [1] 4.2
##
## $b
## [1] NA
##
## $c
## [1] 3.7
##
## $d
## [1] NA
##
## $e
## [1] NA
```

```
lapply(myList, FUN = mean, na.rm = TRUE)
```

```
## $a
## [1] 4.2
##
## $b
## [1] 3.888889
##
## $c
## [1] 3.7
##
## $d
## [1] 3.125
##
## $e
## [1] 2.666667
```

Para mayor legibilidad o si se debemos realizar varias operaciones dentro del argumento FUN, podemos usar el siguiente script:

```
lapply(myList, FUN = function(i){
  mean(i, na.rm = TRUE)
})
```

```
## $a
## [1] 4.2
##
## $b
## [1] 3.888889
##
## $c
## [1] 3.7
```



```
##
## $d
## [1] 3.125
##
## $e
## [1] 2.666667
```

Por ejemplo, si queremos obtener  $i^2$  si el promedio es mayor que 3, y  $i^3$  de lo contrario:

```
lapply(myList, FUN = function(i){
  m <- mean(i, na.rm = TRUE)
  if(m > 3){
    return(i^2)
  }else{
    return(i^3)
  }
})
```

```
## $a
## [1]  4 16 25 25 25  9 25 16 25 16
##
## $b
## [1]  9 25 NA 16 16  4 25 25  4 25
##
## $c
## [1] 25  4 25  9  9  4  4 25 25 25
##
## $d
## [1]  9 NA  4  4 25 25  1 16  9 NA
##
## $e
## [1] 64  8  8 125  8 NA  8 64  8  1
```

### 11.7.3 sapply

La función `sapply()` es una versión modificada de la función `lapply()` que realiza la misma operación pero devuelve el resultado en un formato simplificado siempre que sea posible.

```
lapply(myList, FUN = function(i){
  mean(i, na.rm = TRUE)
})
```

```
## $a
## [1] 4.2
##
## $b
## [1] 3.888889
##
## $c
## [1] 3.7
##
## $d
## [1] 3.125
```

```
##
## $e
## [1] 2.666667
```

```
sapply(myList, FUN = function(i){
  mean(i, na.rm = TRUE)
})
```

```
##      a      b      c      d      e
## 4.200000 3.888889 3.700000 3.125000 2.666667
```

La función `sapply()` es interesante para recuperar, por ejemplo, el elemento “n” de cada elemento de una list. La función que se llama para hacer esto es `'[['`.

```
sapply(myList, FUN = '['[, 2])
```

```
## a b c d e
## 4 5 2 NA 2
```

### 11.7.4 tapply

La función `tapply()` permite aplicar una función tomando como elemento para iterar una variable existente. Imaginamos información sobre especies representadas por letras mayúsculas (por ejemplo, A, B, C) y valores de mediciones biológicas en diferentes ubicaciones.

```
species <- sample(LETTERS[1:10], size = 1000, replace = TRUE)
perf1 <- rnorm(n = 1000, mean = 10, sd = 0.5)
perf2 <- rlnorm(n = 1000, meanlog = 10, sdlog = 0.5)
perf3 <- rgamma(n = 1000, shape = 10, rate = 0.5)
dfSpecies <- data.frame(species, perf1, perf2, perf3)
print(head(dfSpecies, n = 10))
```

```
##   species    perf1    perf2    perf3
## 1      A 11.148355 15327.729 26.02575
## 2      G  9.734033 21394.444 18.81282
## 3      G  9.777218 22716.770 13.59781
## 4      E 10.890801 26037.797 25.61112
## 5      J  9.648831 40798.109 15.98179
## 6      I  9.784595 22456.649 20.40941
## 7      B 10.571920  8462.618 11.79735
## 8      H 10.437621 29347.028 26.93568
## 9      F 10.838245 12882.271 19.42126
## 10     E 10.348212 25829.856 12.52852
```

Podemos obtener fácilmente un resumen de las mediciones para cada especie con la función `tapply()` y la función `summary()`.

```
tapply(dfSpecies$perf1, INDEX = dfSpecies$species, FUN = summary)
```

```
## $A
##   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##  8.431   9.745   9.997  10.002  10.331  11.210
##
```

```
## $B
##      Min. 1st Qu.  Median      Mean 3rd Qu.      Max.
##  8.751   9.884  10.215   10.162  10.504   11.420
##
## $C
##      Min. 1st Qu.  Median      Mean 3rd Qu.      Max.
##  9.062   9.669  10.017   10.027  10.326   11.553
##
## $D
##      Min. 1st Qu.  Median      Mean 3rd Qu.      Max.
##  8.755   9.646   9.983   9.950  10.239   11.068
##
## $E
##      Min. 1st Qu.  Median      Mean 3rd Qu.      Max.
##  8.803   9.636   9.937   9.940  10.307   11.123
##
## $F
##      Min. 1st Qu.  Median      Mean 3rd Qu.      Max.
##  8.849   9.689  10.019  10.003  10.348   11.253
##
## $G
##      Min. 1st Qu.  Median      Mean 3rd Qu.      Max.
##  8.819   9.646   9.947   9.964  10.209   11.146
##
## $H
##      Min. 1st Qu.  Median      Mean 3rd Qu.      Max.
##  8.444   9.445   9.949   9.877  10.241   11.431
##
## $I
##      Min. 1st Qu.  Median      Mean 3rd Qu.      Max.
##  8.395   9.699   9.987   9.956  10.174   11.409
##
## $J
##      Min. 1st Qu.  Median      Mean 3rd Qu.      Max.
##  8.660   9.465   9.824   9.880  10.236   11.169
```

También podemos obtener el valor promedio de cada mediciones combinando una función `sapply()` con la función `tapply()` y usando la función `mean()`.

```
sapply(2:4, FUN = function(i){
  tapply(dfSpecies[,i], INDEX = dfSpecies$species, FUN = mean)
})
```

```
##           [,1]      [,2]      [,3]
## A 10.002105 25782.86 20.61024
## B 10.161999 27095.11 20.44876
## C 10.026614 24723.48 19.85151
## D  9.949698 24434.86 19.83905
## E  9.939783 22772.05 19.20053
## F 10.002797 26707.99 19.18816
## G  9.963703 24426.39 20.20499
## H  9.877476 24288.66 19.59506
## I  9.956013 24307.37 18.99487
## J  9.880460 26842.73 19.06255
```

### 11.7.5 mapply

La función `mapply()` es una versión de la función `sapply()` que usa múltiples argumentos. Por ejemplo, si tenemos una lista de dos elementos `1:5` y `5:1` y queremos agregar 10 al primer elemento y 100 al segundo elemento:

```
mapply(FUN = function(i, j){i+j}, i = list(1:5, 5:1), j = c(10, 100))
```

```
##      [,1] [,2]  
## [1,]   11  105  
## [2,]   12  104  
## [3,]   13  103  
## [4,]   14  102  
## [5,]   15  101
```

## 11.8 Conclusión

Felicitaciones, hemos llegado al final de este capítulo sobre algoritmos. Recordemos este mensaje clave: cuando una operación debe realizarse más de dos veces en un script y repetir el código que ya se ha escrito, es un signo que nos debe llevar a utilizar un bucle. Sin embargo, siempre que sea posible, se recomienda no usar los bucles tradicionales `for()`, `while()`, y `repeat()`, sino preferir operaciones sobre vectores o bucles de la familia `apply`. Esto puede ser difícil de integrar al principio, pero veremos que nuestros scripts serán más fáciles de mantener y leer, y mucho más eficientes si seguimos estos hábitos.

## Chapter 12

# Gestión de proyectos con R

Ahora que hemos visto los conceptos básicos de R, todavía tenemos que abordar un elemento decisivo para el buen funcionamiento de nuestras actividades científicas con R: gestión de proyectos. Consiste en integrar sus desarrollos en un entorno y con una lógica orientada a facilitar su trabajo y aumentar así su eficiencia. Esta es solo una manera de hacer dentro de las infinitas posibilidades, a adaptar para todos y cada uno.

### 12.1 Gestionando archivos y directorios de trabajo.

Entre los archivos de entrada (es decir, los archivos que contienen nuestros datos en bruto), los archivos de salida (por ejemplo, con la función `write()`), los gráficos (siguiente capítulo) y los muchos scripts asociados con un proyecto, se necesita un mínimo de organización para ser eficaz y reanudar rápidamente su proyecto. La solución más sencilla es estructurar su entorno de trabajo en carpetas según cada categoría de archivo. Por ejemplo, con una carpeta “myProject” para el proyecto, que contiene las carpetas “myFiles” para los archivos de entrada, una carpeta “myScripts” para el script R y una carpeta “myOutputs” para los archivos salida (por ejemplo, gráficos y análisis).

```
## -myProject
## |-myFiles
## |-|-data01.csv
## |-|-data02.csv
## |-myScripts
## |-|-myFirstScript.R
## |-myOutputs
## |-|-dataOut01.csv
## |-|-figure01.pdf
```

### 12.2 Gestión de versiones de script

El trabajo en un script es iterativo: incluso si los objetivos se definen desde el principio, volveremos a trabajar algunas partes para obtener, por ejemplo, información adicional, o para optimizar esta o aquella función, o hacer un script generalizable para comunicarlo a la comunidad científica o simplemente un colega. A veces, lo que veremos como una mejora finalmente será un error, y volver al estado inicial puede ser difícil. Así que tenemos que gestionar las versiones.

En la mayoría de los laboratorios hay servicios de control de versiones, el más conocido es GIT (<https://git-scm.com/>) y Subversion (<https://subversion.apache.org/>). Cuando GIT o Subversion están disponibles, se recomienda usarlos. Si no tenemos acceso a estos servicios, hay servicios en línea gratuitos como GitHub (<https://github.com/>; este libro utiliza GitHub). Hay muchas otras soluciones como GitLab (<https://about.gitlab.com/>), Bitbucket (<https://bitbucket>).

org/), SourceForge (<https://sourceforge.net/>), GitKraken (<https://gitkraken.com/>), o Launchpad (<https://launchpad.net/>).

El uso de estos diferentes servicios de versiones está fuera del alcance de este libro. Para los principiantes o para proyectos que no requieren trabajo colaborativo en scripts, una alternativa es administrar sus versiones manualmente. Por ejemplo, una solución es agregar un número al final de su nombre de archivo de script (por ejemplo, “myFirstScript\_01.R”). Tan pronto como se realice una modificación importante en este script, bastará con guardarlo con un nuevo nombre (p. Ej., “MyFirstScript\_02.R”) y colocar el script antiguo en una carpeta aparte para no desordenar el espacio de trabajo y hacer errores de version. En caso de problemas, podemos volver fácilmente al script anterior y reanudar nuestro trabajo.

```
## -myProject
## |-myFiles
## |--data01.csv
## |--data02.csv
## |-myScripts
## |--myFirstScript04.R
## |--ARCHIVES
## |--|-myFirstScript01.R
## |--|-myFirstScript02.R
## |--|-myFirstScript03.R
## |-myOutputs
## |--dataOut01.csv
## |--figure01.pdf
```

## 12.3 Gestion de documentacion

La documentación de su código es esencial para volver fácilmente al trabajo o comunicar su trabajo con colegas y la comunidad científica. Un código bien documentado será comprensible por un número mayor y, por lo tanto, se utilizará más. Por eso es importante adoptar buenas técnicas y practicas.

Ya hemos visto que hay varias formas de escribir su código con R porque es un lenguaje bastante permisivo. El primer paso hacia un código legible y reproducible es adoptar un estilo de código claro y coherente y ... ¡hecho para humanos! Porque incluso si nuestro código está destinado a ser ejecutado por máquinas, debe seguir siendo comprensible de todas las personas que lo consultarán. Es por ejemplo poner espacios después de las comas, o usar la indentación. Por supuesto, la legibilidad del código debe equilibrarse con la optimización del código para grandes conjuntos de datos, pero en la mayoría de los casos podemos asociar un código claro y optimizado. Entonces, el primer paso de la documentación y su administración es escribir primero su código pensando en las personas que lo leerán y lo reproducirán.

El segundo paso es comentar su código. Los comentarios son esenciales cuando privilegiamos el código optimizado para el rendimiento pero que pierde en legibilidad. Los comentarios son superfluos si el código está bien escrito y los objetos y funciones están bien nombrados. Esto significa que los comentarios no deben usarse para explicar un código mal escrito, sino que desde el principio escribir bien nuestro código. Los comentarios son útiles para proporcionar elementos contextuales (por ejemplo, la elección de un método sobre otro en la literatura). El lugar de los comentarios puede estar al final de las líneas o en líneas separadas.

Para un proyecto pequeño en R es esencial que cada script comience con una descripción de sus contenidos para que podamos saber rápidamente de qué se trata. Eso es lo que hicimos al principio de este libro:

```
# -----
# Aquí hay un script para adquirir los conceptos básicos
# con R
# fecha de creación : 25/06/2018
# autor : François Rebaudo
# -----
```

```
# [1] Creación del objeto número de repeticiones.
# -----

nbrRep <- 5

# [2] calculos simples
# -----

pi * nbrRep^2
```

Aquí los comentarios que siguen al encabezado no son necesarios porque el nombre del objeto se entiende por sí mismo. Nuestro archivo se convierte en:

```
# -----
# Aquí hay un script para adquirir los conceptos básicos
# con R
# fecha de creación : 25/06/2018
# autor : François Rebaudo
# -----

nbrRep <- 5
pi * nbrRep^2
```

Para un proyecto grande con muchas funciones para ser utilizado por otros usuarios, es preferible que la documentación del código esté separada, en un archivo de ayuda específico. Este es el caso de todos los paquetes R! Para gestionar la documentación de un paquete (y por lo tanto de todas las funciones), de nuevo hay muchas posibilidades. Lo más común es usar el paquete R roxygen2. Sin entrar en detalles, aquí hay algunos ejemplos de la documentación del paquete.

```
#' Add together two numbers
#'
#' @param x A number
#' @param y A number
#' @return The sum of \code{x} and \code{y}
#' @examples
#' add(1, 1)
#' add(10, 1)
add <- function(x, y) {
  x + y
}
```

```
#' Sum of vector elements.
#'
#' `sum` returns the sum of all the values present in its arguments.
#'
#' This is a generic function: methods can be defined for it directly
#' or via the [Summary()] group generic. For this to work properly,
#' the arguments `...` should be unnamed, and dispatch is on the
#' first argument.
sum <- function(..., na.rm = TRUE) {}
```

Esto nos permite escribir la documentación de cada función junto a la función. El paquete roxygen2 generará a partir de estos comentarios un documento de ayuda accesible con la función '?'. A menos que escribamos un nuevo paquete, los comentarios

simples serán suficientes, y el desarrollo de un paquete está fuera del alcance de este libro.

## 12.4 Conclusión

Felicitaciones. Este capítulo marca el final de la primera parte de este libro. Ahora tenemos lo básico para llevar a cabo nuestros proyectos con R. En la siguiente parte veremos los gráficos y cómo hacer figuras en el marco de los artículos científicos.



## **Part II**

# **Estudio de caso**



## Chapter 13

# Analizar datos de loggers de temperatura

En estudios de biología, ecología, o agronomía, frecuentemente usamos datos de temperatura de dataloggers. En este estudio vamos a ver como analizar esos datos usando datos de temperatura del altiplano Boliviano cerca de la ciudad de El Alto. El primer paso es transformar los datos del datalogger en un formato que sea fácil de leer para R. Usaremos un archivo CSV y la función `read.table()`. El archivo se puede descargar desde el sitio web del libro en GitHub ([https://github.com/frareb/myRBook\\_SP/blob/master/myFiles/E05C13.csv](https://github.com/frareb/myRBook_SP/blob/master/myFiles/E05C13.csv) ; clic derecho sobre el enlace y seleccionar "Guardar destino como").

```
bdd <- read.table("myFiles/E05C13.csv", skip = 1, header = TRUE,
  sep = ",", dec = ".", stringsAsFactors = FALSE)
colnames(bdd) <- c("id", "date", "temp")
head(bdd)
```

```
##   id           date temp
## 1  1 11/12/15 23:00:00 4.973
## 2  2 11/12/15 23:30:00 4.766
## 3  3 11/13/15 00:00:00 4.844
## 4  4 11/13/15 00:30:00 4.844
## 5  5 11/13/15 01:00:00 5.076
## 6  6 11/13/15 01:30:00 5.282
```

```
tail(bdd)
```

```
##           id           date temp
## 32781 32781 09/25/17 21:00:00 7.091
## 32782 32782 09/25/17 21:30:00 6.914
## 32783 32783 09/25/17 22:00:00 6.813
## 32784 32784 09/25/17 22:30:00 6.611
## 32785 32785 09/25/17 23:00:00 6.331
## 32786 32786 09/25/17 23:30:00 5.385
```

```
str(bdd)
```

```
## 'data.frame':   32786 obs. of  3 variables:
## $ id : int  1 2 3 4 5 6 7 8 9 10 ...
## $ date: chr  "11/12/15 23:00:00" "11/12/15 23:30:00" "11/13/15 00:00:00" "11/13/15 00:30:00" ...
## $ temp: num  4.97 4.77 4.84 4.84 5.08 ...
```

Podemos observar que la fecha esta al formato `character`, y que contiene la fecha con el mes, el día, y el año separados con `/`, un espacio, y la hora con horas de 0 a 24, minutos, y segundos, separados con `:` (ejemplo: 11/12/15 23:00:00 para el

12 de Noviembre de 2015 a las 11 de la noche). Vamos a separar la información en varios objetos. Primero vamos a separar la fecha de la hora. Para esto vamos a usar la función `strsplit()` usando como separador el espacio entre la fecha y la hora.

```
strsplit("11/12/15 23:00:00", split = " ")
```

```
## [[1]]
## [1] "11/12/15" "23:00:00"
```

Como indican los corchetes dobles, la función devuelve un objeto en el formato `list`. Nosotros queremos el vector que corresponde al primer elemento de la `list` entonces vamos a añadir `[[1]]`.

```
strsplit("11/12/15 23:00:00", split = " ")[[1]]
```

```
## [1] "11/12/15" "23:00:00"
```

El primer elemento del vector es la fecha. Para tener todas las fechas vamos a hacer un bucle con la función `sapply()`.

```
bddDay <- sapply(strsplit(bdd[, 2], split = " "), "[", 1)
head(bddDay)
```

```
## [1] "11/12/15" "11/12/15" "11/13/15" "11/13/15" "11/13/15" "11/13/15"
```

A continuación vamos a necesitar las fechas en el formato `factor` (función `aggregate()` para sacar la información por día). Entonces necesitamos transformar el objeto en el formato `factor` con la función `as.factor()`.

```
bddDay <- as.factor(sapply(strsplit(bdd[, 2], split = " "), "[", 1))
head(bddDay)
```

```
## [1] 11/12/15 11/12/15 11/13/15 11/13/15 11/13/15 11/13/15
## 684 Levels: 01/01/16 01/01/17 01/02/16 01/02/17 01/03/16 ... 12/31/16
```

Haciendo la transformación hacia el formato `factor`, los niveles de nuestro objeto están en orden alfabético como si las fechas estuvieran texto. Nosotros queremos ordenar las fechas usando las fechas. Para esto vamos a hacer un vector con todas las fechas únicas con la función `unique()`, y después ordenar las fechas con la función `sort.list()` usando las fechas con la función `as.POSIXct()`. Vamos a usar el vector llamado `lev` para especificar como deben estar los niveles de `factor` de nuestras fechas.

```
bddDay <- as.factor(sapply(strsplit(bdd[, 2], split = " "), "[", 1))
udate <- unique(bddDay)
lev <- udate[sort.list(as.POSIXct(strptime(udate, "%m/%d/%y")))]
bddDay <- factor(bddDay, levels = lev)
head(bddDay)
```

```
## [1] 11/12/15 11/12/15 11/13/15 11/13/15 11/13/15 11/13/15
## 684 Levels: 11/12/15 11/13/15 11/14/15 11/15/15 11/16/15 ... 09/25/17
```

Ahora podemos añadir las fechas como nueva columna del objeto `bdd` y hacer lo mismo para las horas (no hay que reordenar los niveles de hora ya que el orden por defecto corresponde a lo que queremos).

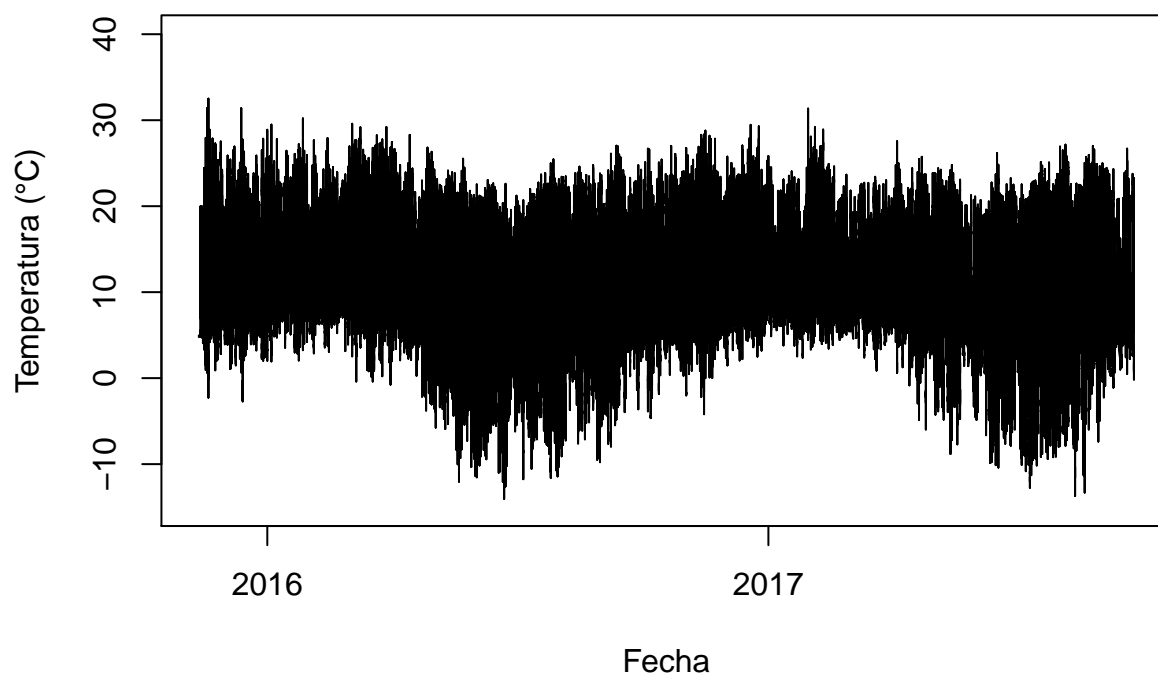
```
bdd$bddDay <- bddDay
bdd$bddHour <- as.factor(sapply(strsplit(bdd[, 2], split = " "), "[", 2))
head(bdd)
```

```
##   id          date temp  bddDay bddHour
## 1   1 11/12/15 23:00:00 4.973 11/12/15 23:00:00
```

```
## 2 2 11/12/15 23:30:00 4.766 11/12/15 23:30:00
## 3 3 11/13/15 00:00:00 4.844 11/13/15 00:00:00
## 4 4 11/13/15 00:30:00 4.844 11/13/15 00:30:00
## 5 5 11/13/15 01:00:00 5.076 11/13/15 01:00:00
## 6 6 11/13/15 01:30:00 5.282 11/13/15 01:30:00
```

Podemos visualizar los datos con la función `plot()`, especificando el formato de las fechas con la función `as.Date()`

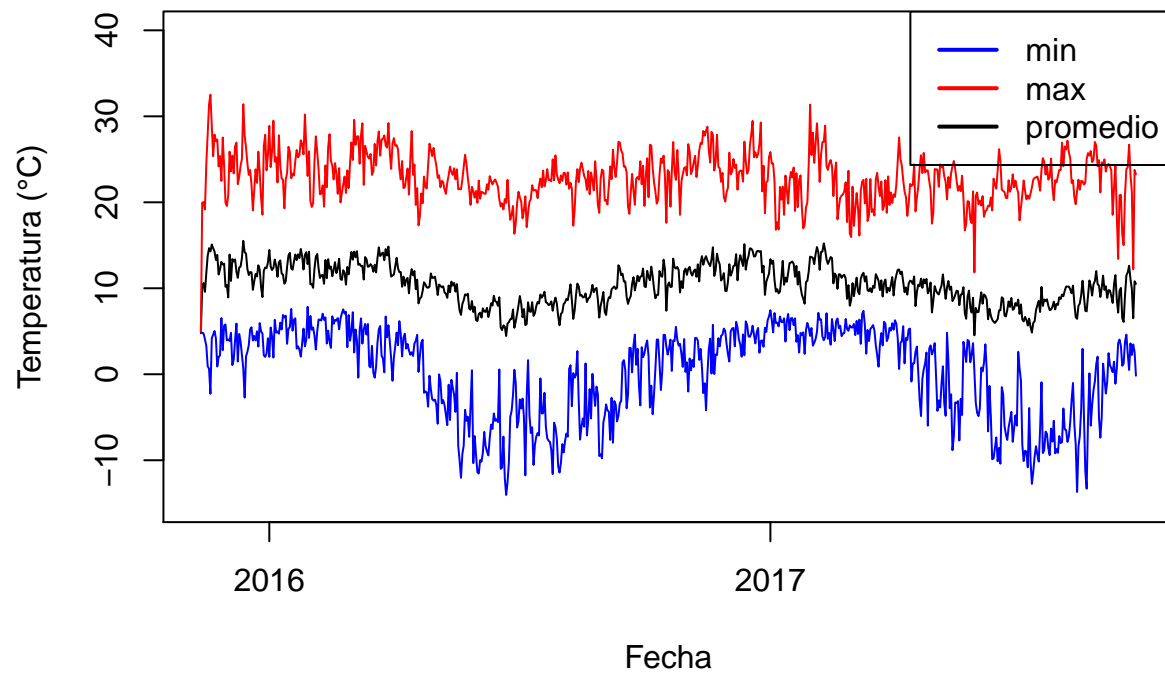
```
plot(x = as.Date(bdd$bddDay, format = "%m/%d/%y"), y = bdd$temp,
     type = 'l', ylim = c(-15, 40),
     xlab = "Fecha", ylab = "Temperatura (°C)")
```



Podemos simplificar la información calculando únicamente las temperaturas mínimas, promedias, y máximas con la función `aggregate()`.

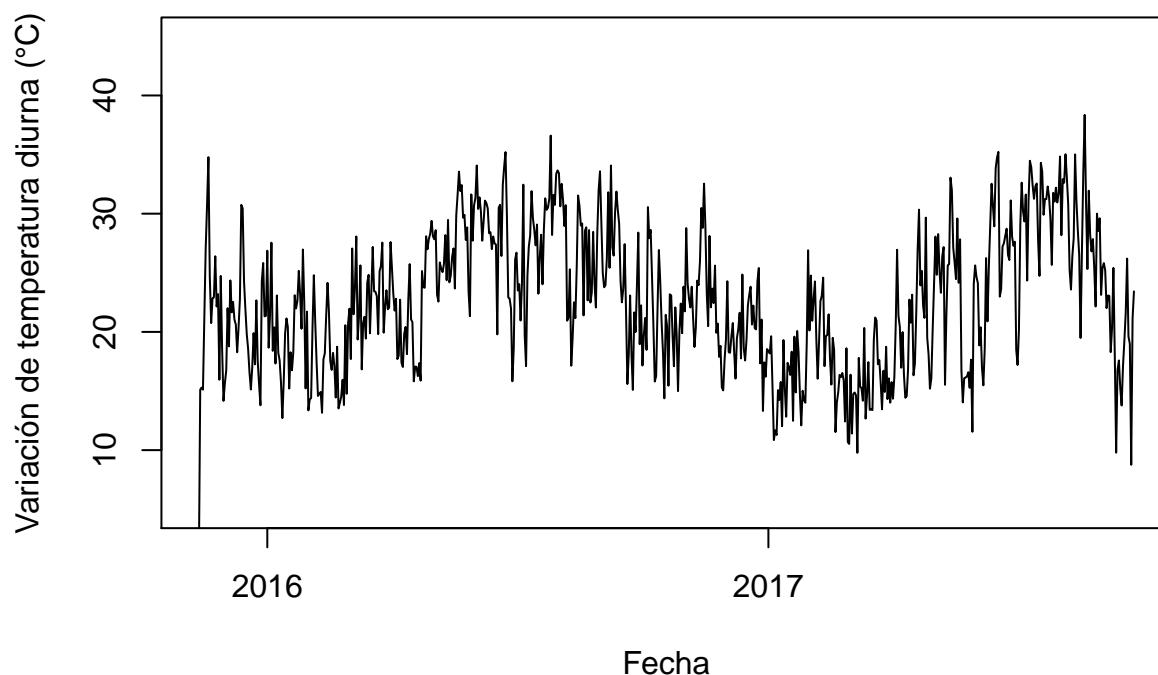
```
tempDayMean <- aggregate(x = bdd[, 3], by = list(bdd[, 4]), FUN = mean)
tempDayMin <- aggregate(x = bdd[, 3], by = list(bdd[, 4]), FUN = min)
tempDayMax <- aggregate(x = bdd[, 3], by = list(bdd[, 4]), FUN = max)
plot(x = as.Date(tempDayMean[, 1], format = "%m/%d/%y"),
     y = tempDayMean[, 2], type = 'l', ylim = c(-15, 40),
     xlab = "Fecha", ylab = "Temperatura (°C)")
points(x = as.Date(tempDayMin[, 1], format = "%m/%d/%y"),
       y = tempDayMin[, 2], type = 'l', col = 4)
points(x = as.Date(tempDayMax[, 1], format = "%m/%d/%y"),
       y = tempDayMax[, 2], type = 'l', col = 2)
legend("topright", legend = c("min", "max", "promedio"),
```

```
lty = 1, lwd = 2, col = c(4, 2, 1))
```



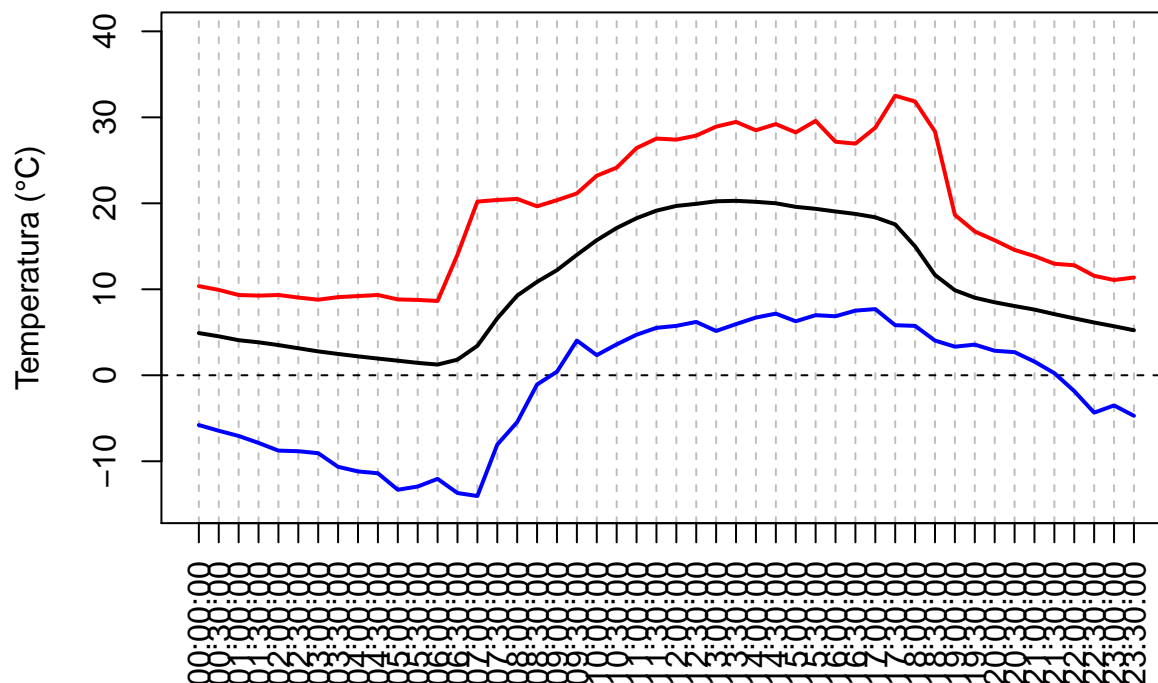
También podemos calcular la diferencia entre la temperatura máxima y la temperatura mínima (variación de temperatura diurna).

```
tempDayTR <- tempDayMax[, 2] - tempDayMin[, 2]
plot(x = as.Date(tempDayMean[, 1], format = "%m/%d/%y"),
     y = tempDayTR, type = 'l', ylim = c(5, 45),
     xlab = "Fecha", ylab = "Variación de temperatura diurna (°C)")
```



Otra posibilidad es de agrupar los datos para tener la temperatura promedio de un día con la función `aggregate()`.

```
tempHourMean <- aggregate(x = bdd[, 3], by = list(bdd[, 5]), FUN = mean)
tempHourMin <- aggregate(x = bdd[, 3], by = list(bdd[, 5]), FUN = min)
tempHourMax <- aggregate(x = bdd[, 3], by = list(bdd[, 5]), FUN = max)
hours <- seq(from = 0, to = 23.5, by = 0.5)
plot(x = hours,
     y = tempHourMean[, 2], type = 'l', ylim = c(-15, 40),
     xlab = "", ylab = "Temperatura (°C)", lwd = 2,
     xaxt = "n", panel.first = {
       abline(v = hours, col = "gray", lty = 2)
       abline(h = 0, lty = 2)
     })
axis(side = 1, at = hours, labels = tempHourMean[, 1], las = 2)
points(x = hours, y = tempHourMin[, 2], type = 'l', col = 4, lwd = 2)
points(x = hours, y = tempHourMax[, 2], type = 'l', col = 2, lwd = 2)
```



También podemos calcular las temperaturas de los días para cada mes.

```
meses <- c("Enero", "Febrero", "Marzo", "Abril", "Mayo", "Junio",
          "Julio", "Agosto", "Septiembre", "Octubre", "Noviembre", "Diciembre")
hours <- seq(from = 0, to = 23.5, by = 0.5)
bddMonth <- sapply(strsplit(as.character(bdd$bddDay), split = "/"), "[", 1)
tempDayEachMonth <- lapply(sort(unique(bddMonth)), function(myMonth){
  bddX <- bdd[bddMonth == myMonth, ]
  tempHourMean <- aggregate(x = bddX[, 3], by = list(bddX[, 5]), FUN = mean)
  tempHourMin <- aggregate(x = bddX[, 3], by = list(bddX[, 5]), FUN = min)
  tempHourMax <- aggregate(x = bddX[, 3], by = list(bddX[, 5]), FUN = max)
  return(data.frame(tempHourMean, tempHourMin, tempHourMax))
})

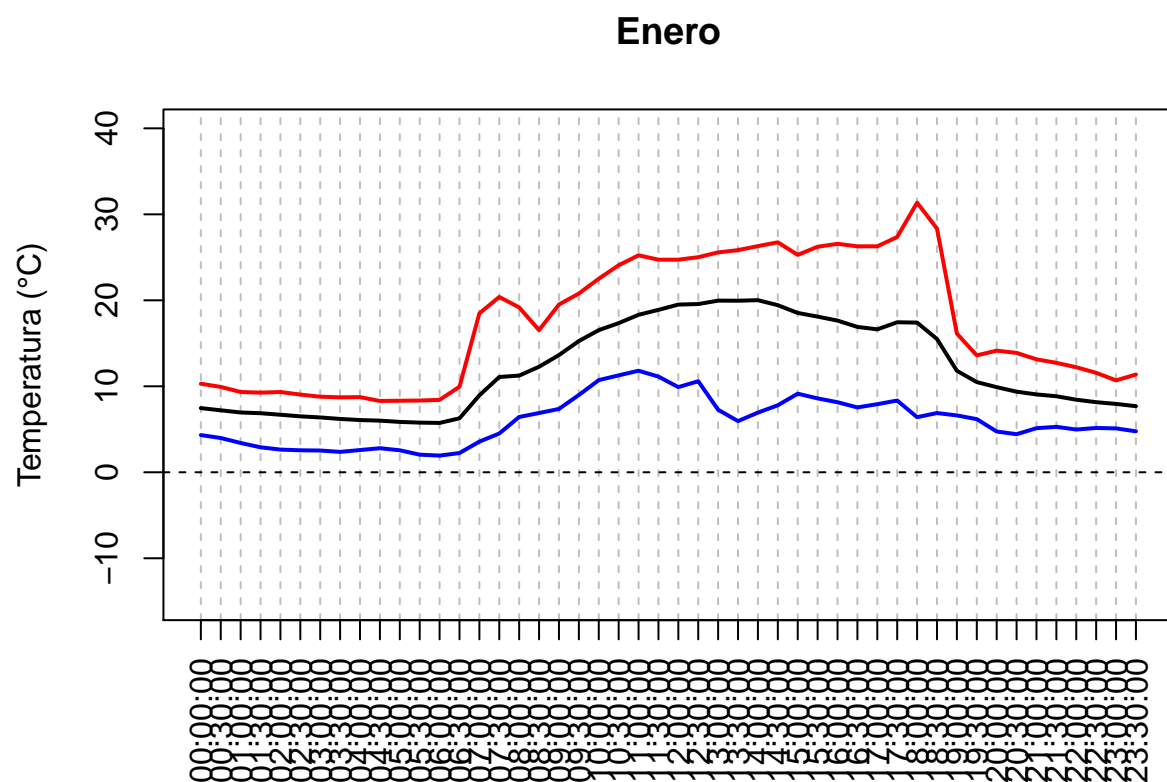
for (i in seq_along(tempDayEachMonth)){
  plot(x = hours, y = tempDayEachMonth[[i]][, 2],
       type = 'l', ylim = c(-15, 40),
       xlab = "", ylab = "Temperatura (°C)", lwd = 2,
       main = meses[i],
       xaxt = "n", panel.first = {
         abline(v = hours, col = "gray", lty = 2)
         abline(h = 0, lty = 2)
       })
  axis(side = 1, at = hours, labels = tempHourMean[, 1], las = 2)
  points(x = hours, y = tempDayEachMonth[[i]][, 4],
        type = 'l', col = 4, lwd = 2)
```



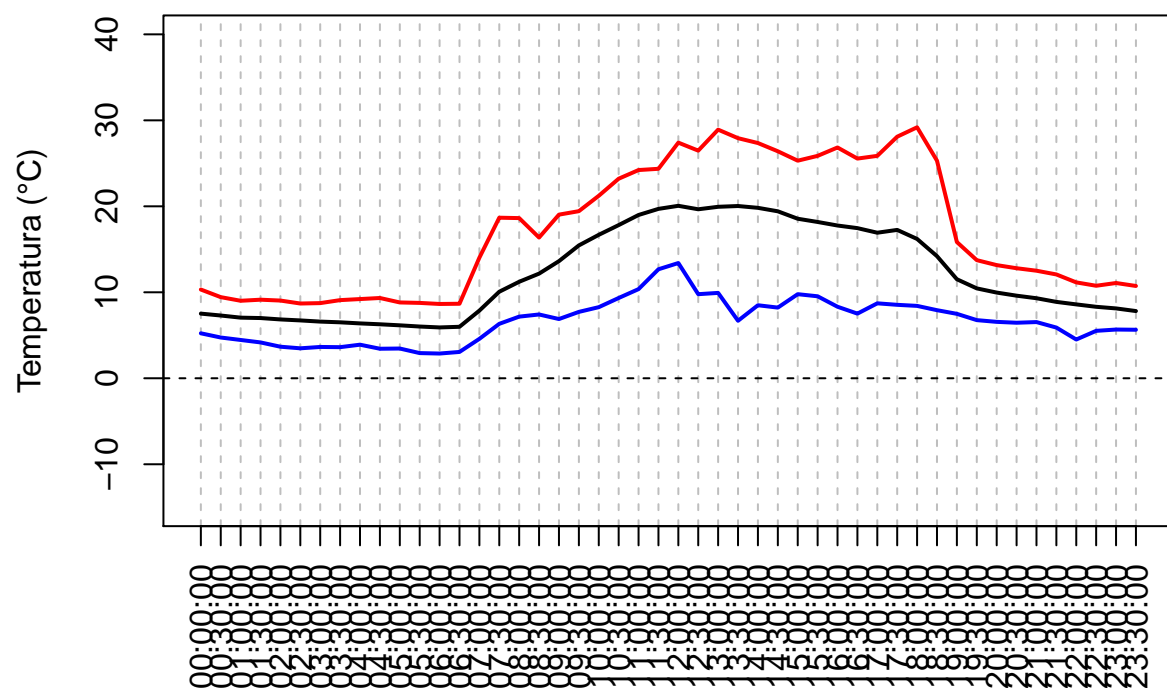
```

points(x = hours, y = tempDayEachMonth[[i]][, 6],
       type = 'l', col = 2, lwd = 2)
}

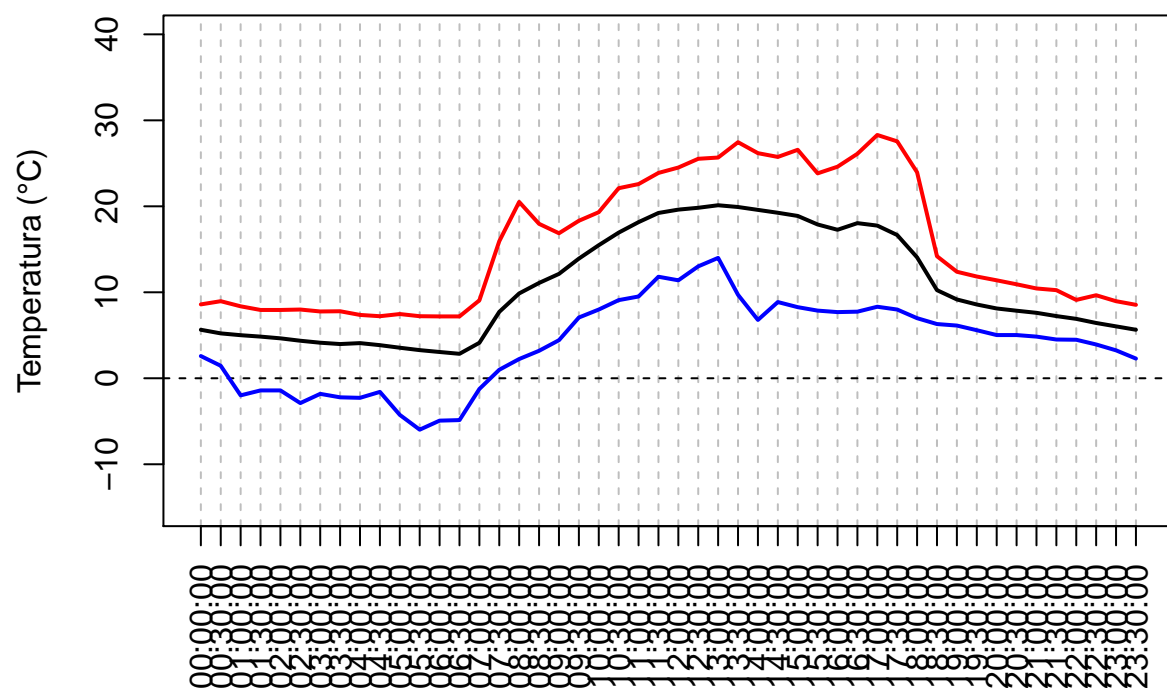
```



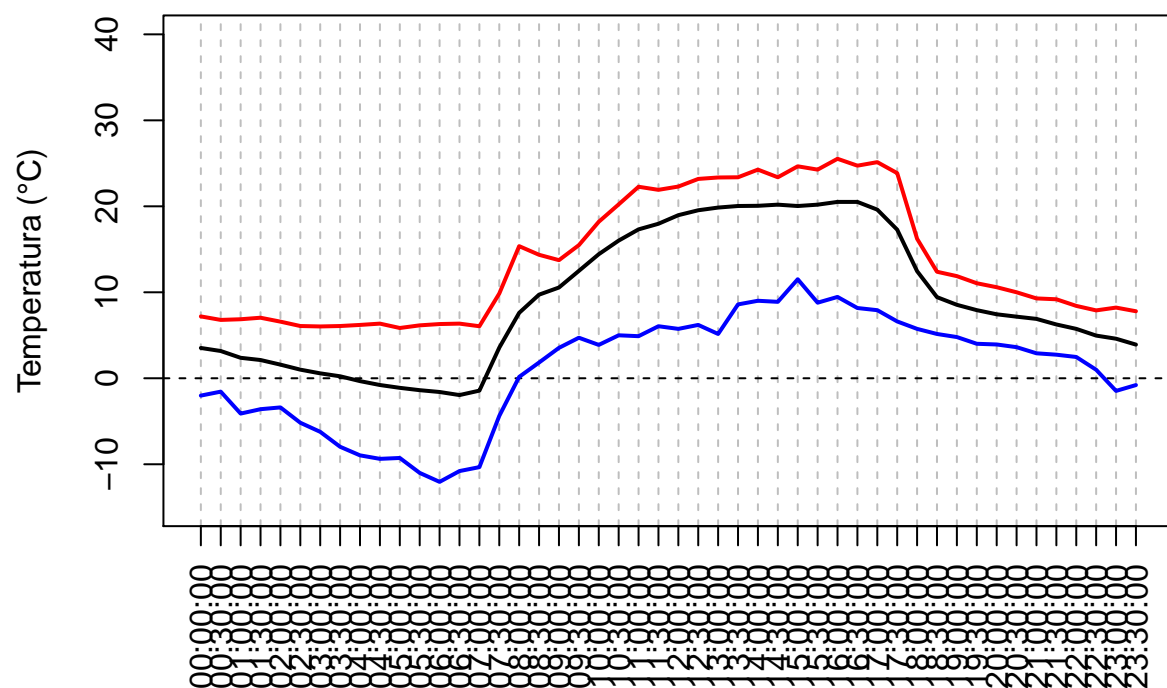
## Febbraio



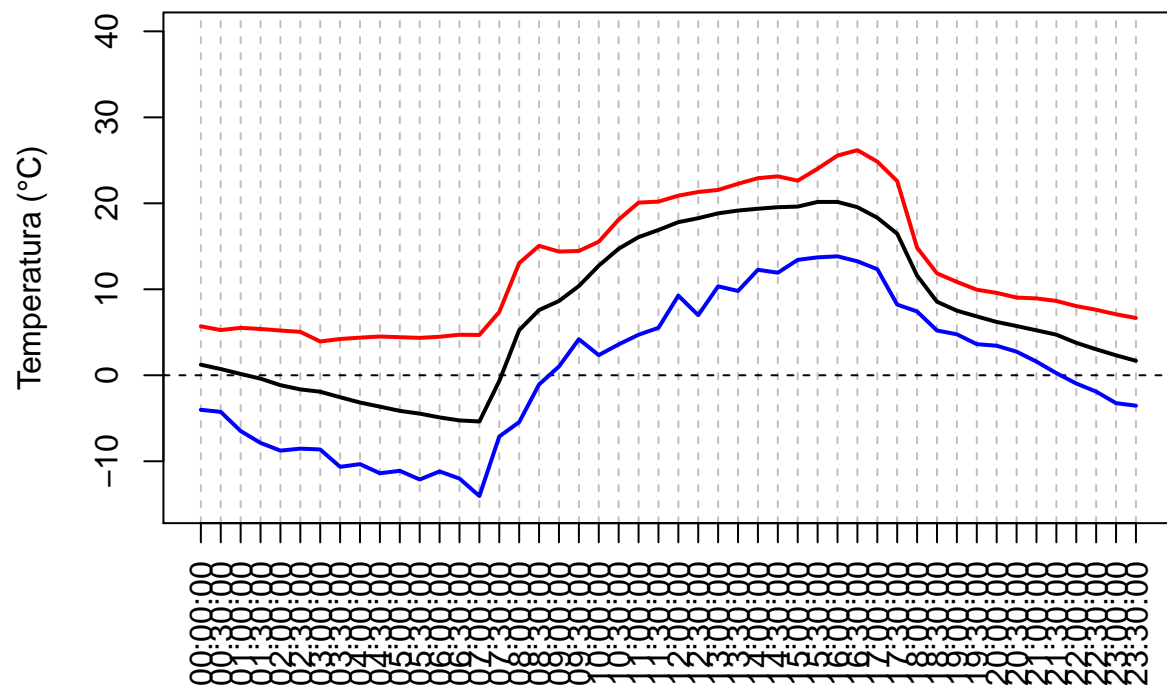


**Abril**

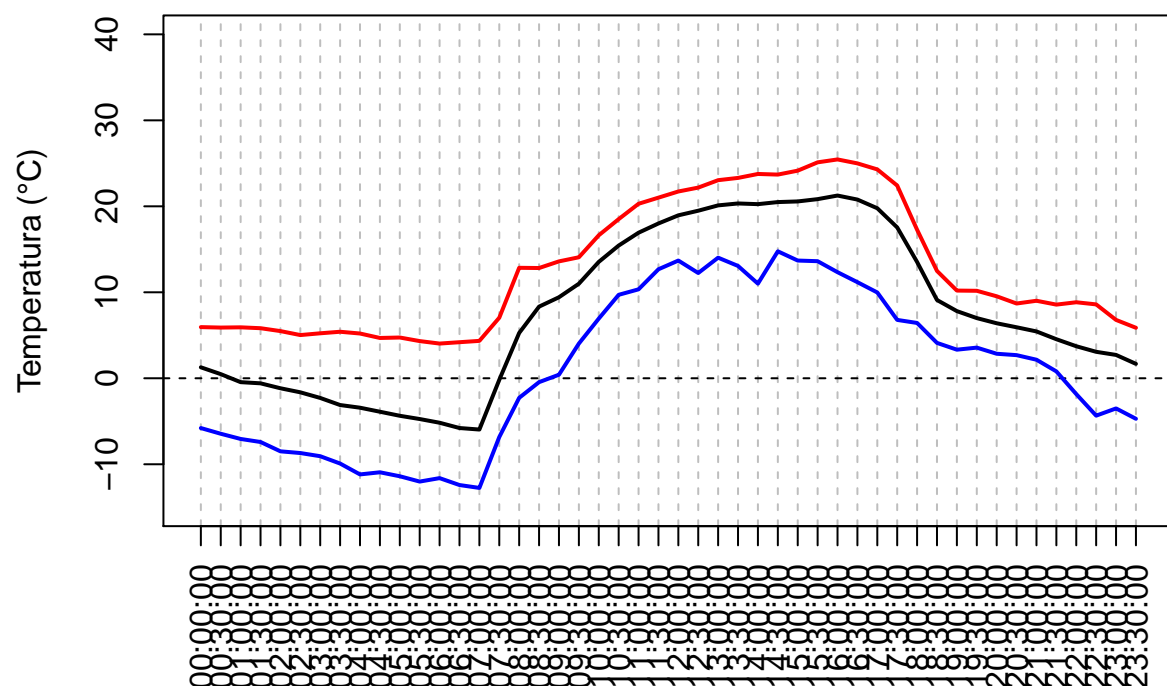
## Mayo



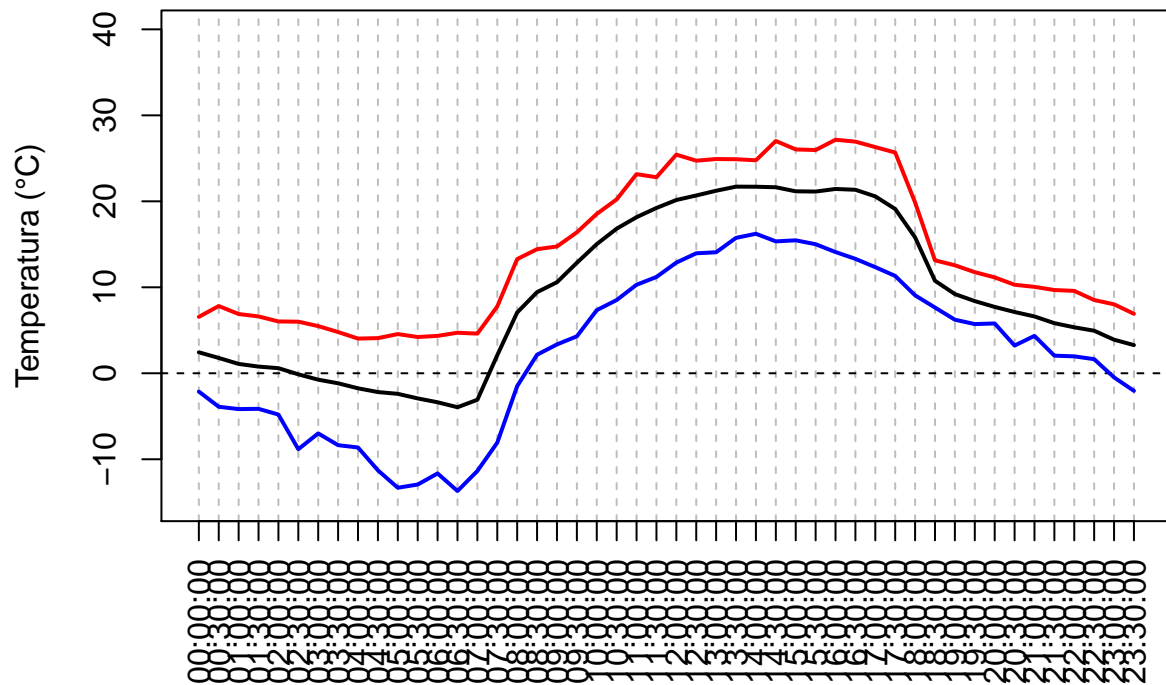
## Junio



# Julio

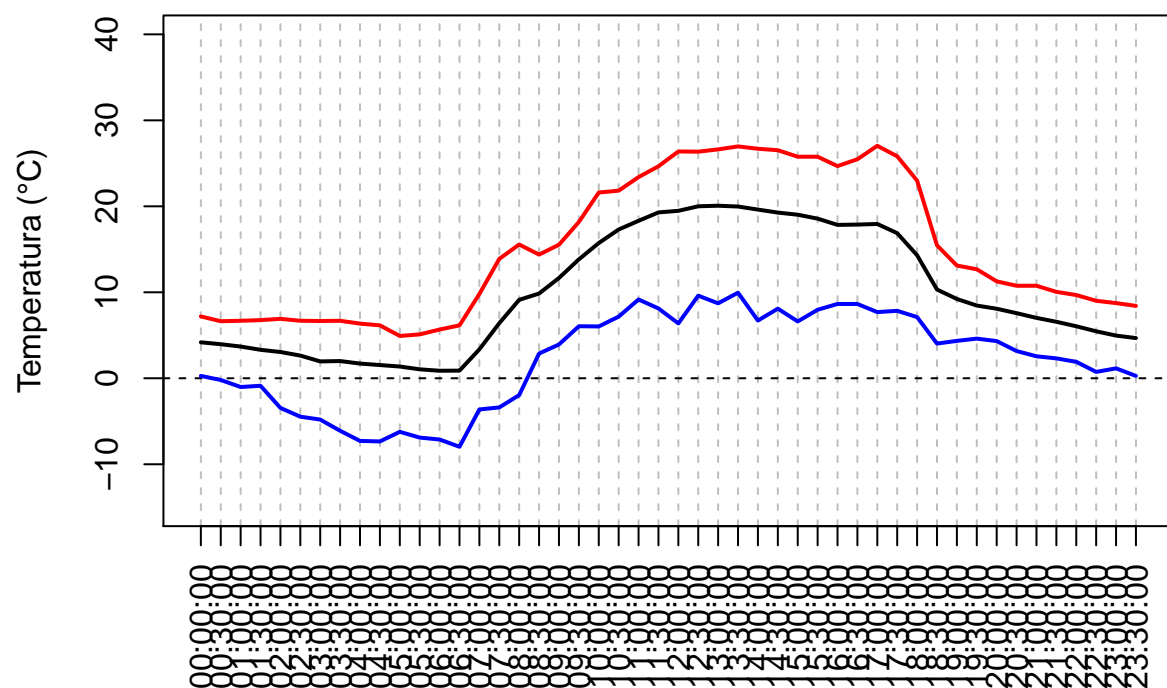


## Agosto

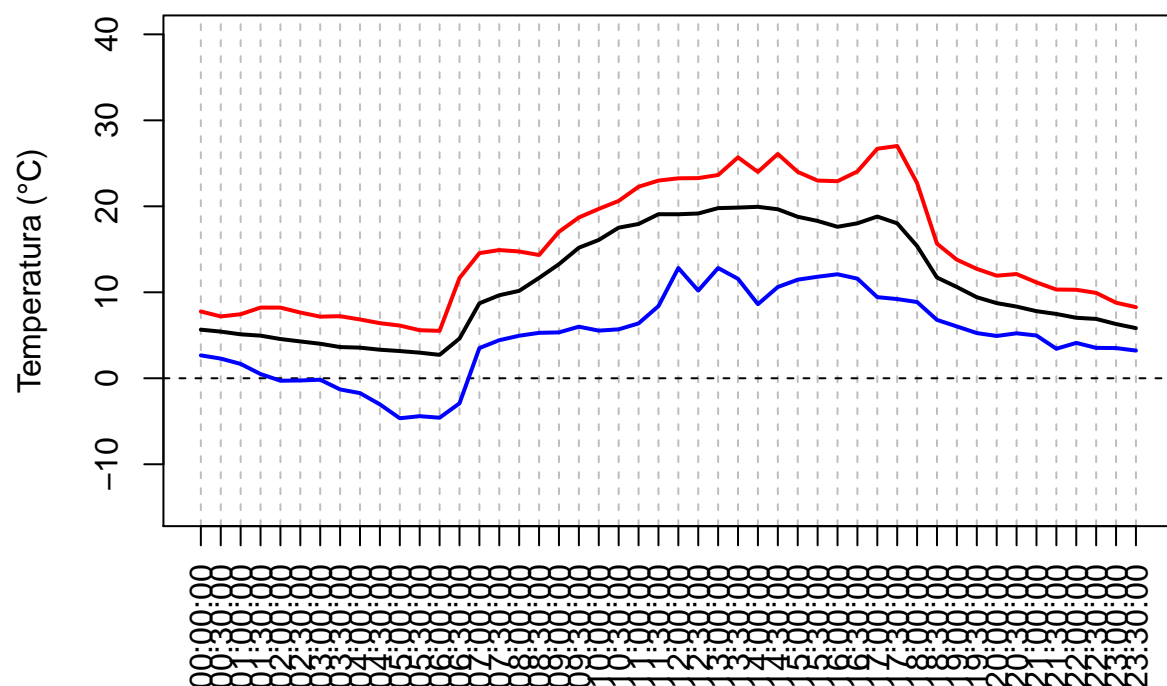




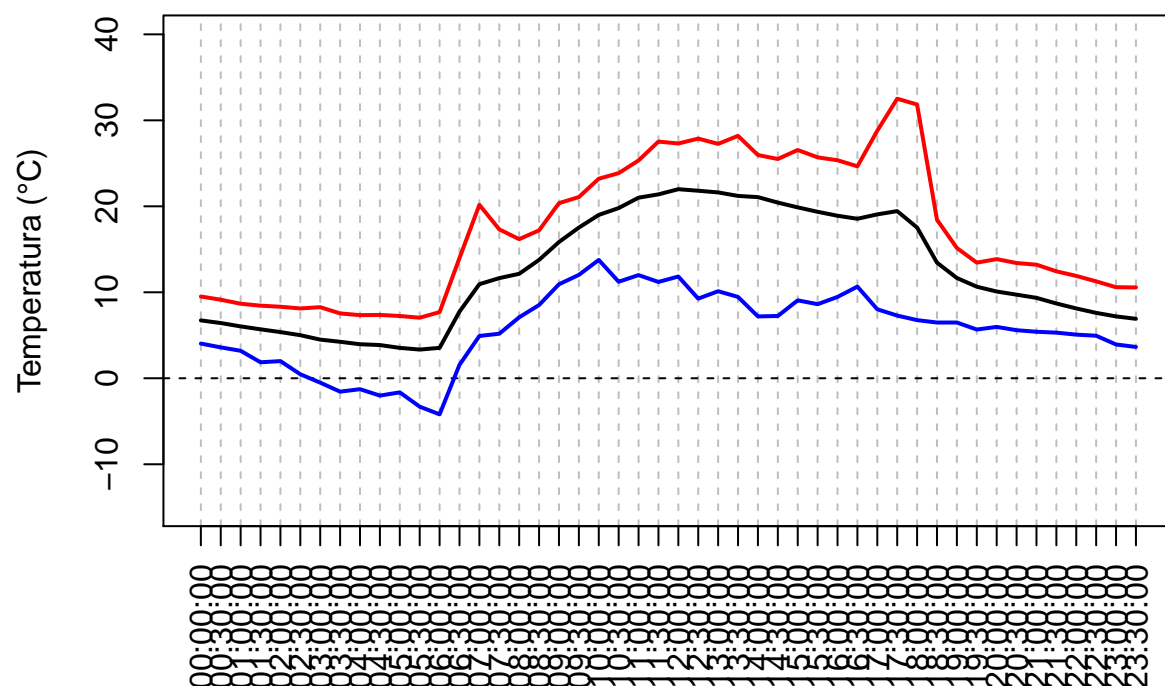
## Septiembre



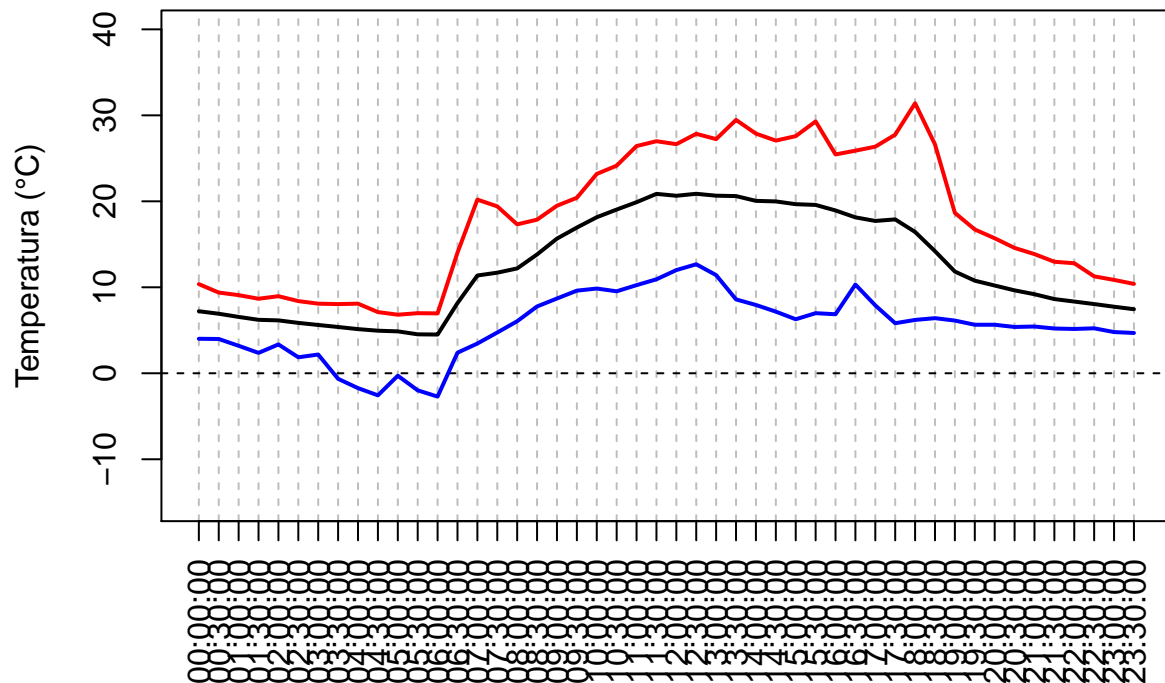
# Octubre



## Noviembre

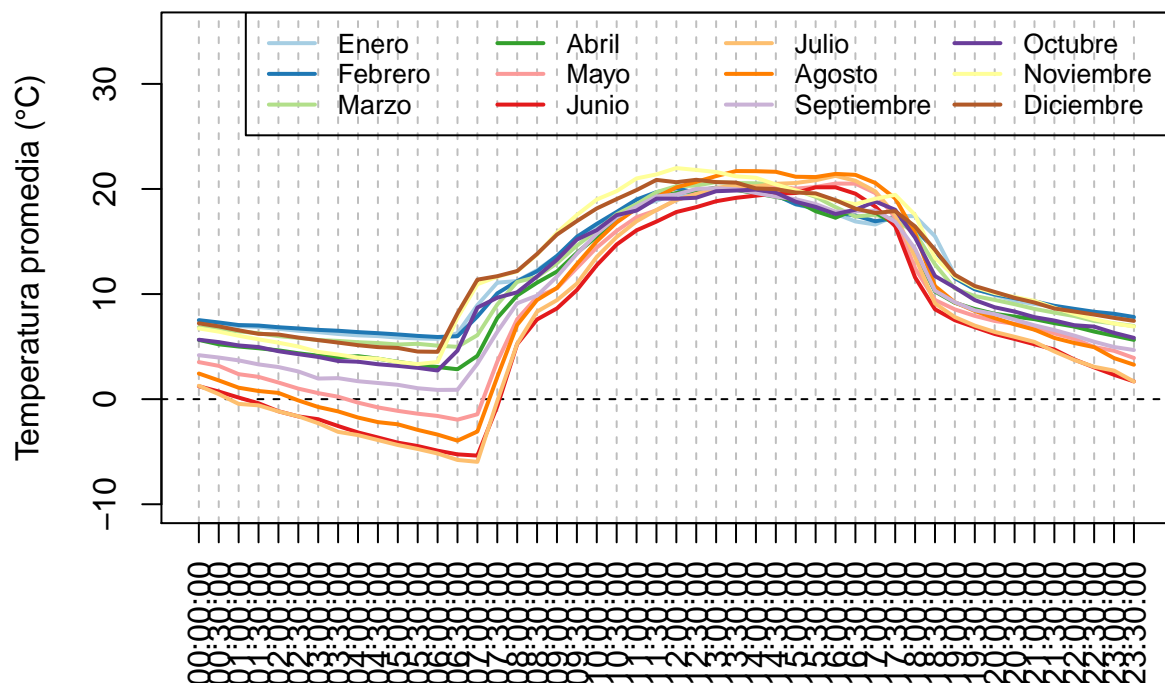


## Diciembre



O todo en un mismo grafico, y la variación de temperatura diurna para cada mes.

```
plot(x = hours, y = tempDayEachMonth[[1]][, 2], type = 'n', ylim = c(-10, 35),
     xlab = "", ylab = "Temperatura promedio (°C)",
     xaxt = "n",
     panel.first = {
       abline(v = hours, col = "gray", lty = 2)
       abline(h = 0, lty = 2)
     })
axis(side = 1, at = hours, labels = tempHourMean[, 1], las = 2)
myColors <- c("#A6CEE3", "#1F78B4", "#B2DF8A", "#33A02C", "#FB9A99",
              "#E31A1C", "#FDBF6F", "#FF7F00", "#CAB2D6", "#6A3D9A", "#FFFF99",
              "#B15928")
for (i in seq_along(tempDayEachMonth)){
  points(x = hours,
         y = tempDayEachMonth[[i]][, 2],
         type = 'l', col = myColors[i], lwd = 2)
}
legend("topright", ncol = 4, legend = meses, col = myColors,
      lty = 1, lwd = 2, cex = 0.8)
```



```
plot(x = hours, y = tempDayEachMonth[[1]][, 2], type = 'n', ylim = c(0, 30),
     xlab = "", ylab = "Variación de temperatura diurna (°C)",
     xaxt = "n",
     panel.first = {
       abline(v = hours, col = "gray", lty = 2)
       abline(h = 0, lty = 2)
     })
axis(side = 1, at = hours, labels = tempHourMean[, 1], las = 2)
myColors <- c("#A6CEE3", "#1F78B4", "#B2DF8A", "#33A02C", "#FB9A99",
              "#E31A1C", "#FDBF6F", "#FF7F00", "#CAB2D6", "#6A3D9A", "#FFFF99",
              "#B15928")
for (i in seq_along(tempDayEachMonth)){
  points(x = hours,
         y = tempDayEachMonth[[i]][, 6] - tempDayEachMonth[[i]][, 4],
         type = 'l', col = myColors[i], lwd = 2)
}
legend("topright", ncol = 4, legend = meses, col = myColors,
      lty = 1, lwd = 2, cex = 0.8)
```

