Aprender R: iniciación y perfeccionamiento

François Rebaudo

2018-07-17

Contents

1 Preámbulo **5**

2 Agradecimientos **7**

3 Licencia **9**

4 Introducción **11**
4.1 ¿Por qué aprender R? . . . 11
4.2 Este libro . . . 11
4.3 Lectura adicional en español . . . 11

I Conceptos básicos **13**

5 Primeros pasos **15**
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

6 Elegir un entorno de desarrollo **27**
6.1 Editores de texto y entorno de desarrollo . . . 27
6.2 RStudio . . . 27
6.3 Notepad++ avec Npp2R . . . 30
6.4 Geany . . . 33
6.5 Otras soluciones . . . 33
6.6 Conclusión . . . 35

7 Tipos de datos **37**
7.1 El tipo `numeric` . . . 37
7.2 El tipo `character` . . . 39
7.3 El tipo `factor` . . . 41
7.4 El tipo `logical` . . . 41
7.5 Acerca de `NA` . . . 42
7.6 Conclusión . . . 43

8 Contenedores de datos **45**
8.1 El contenedor `vector` . . . 45
8.2 El contenedor `list` . . . 52
8.3 El contenedor `data.frame` . . . 64
8.4 El contenedor `matrix` . . . 69
8.5 El contenedor `array` . . . 74
8.6 Conclusión . . . 76

9 Las funciones **77**
9.1 ¿Qué es una función? . . . 77
9.2 Las funciones más comunes . . . 77
9.3 Otras funciones útiles . . . 77
9.4 Escribe una función . . . 77

10 Importar y exportar datos **79**
10.1 Leer datos de un archivo . . . 79
10.2 Guardar datos para R . . . 79
10.3 Exportar datos . . . 79

11 Los bucles **81**
11.1 ¿Por qué hacer bucles? . . . 81
11.2 El bucle `if` . . . 81
11.3 El bucle `switch` . . . 81
11.4 El bucle `for` . . . 81
11.5 El bucle `while` . . . 81
11.6 `repeat`, `next`, `break`, `stop` . . . 81
11.7 Los bucles de la familia `apply` . . . 82

II Los gráficos **83**

12 Gráficos simples **85**
12.1 plot . . . 85
12.2 hist . . . 85
12.3 barplot . . . 85
12.4 boxplot . . . 85
12.5 image y contour . . . 85

13 Gestión del color **87**
13.1 colors() . . . 87
13.2 RGB . . . 87
13.3 Paletas . . . 87

14 Gráficos compuestos **89**
14.1 mfrow . . . 89
14.2 layout . . . 89

15 Manipular gráficos **91**
15.1 Inkscape . . . 91
15.2 The Gimp . . . 91

III Estadísticas con R **93**

16 Estadísticas descriptivas **95**

IV Estudio de caso **97**

17 Analizar datos de loggers de temperatura **99**

Chapter 1

Preámbulo

Este libro está incompleto por el momento y está leyendo su versión preliminar. Si tiene algún comentario, sugerencia o si identifica errores, no dude en enviarme un correo electrónico (`francois.rebaudo@ird.fr`[1]), o si está familiarizado con GitHub en el sitio web del proyecto (`https://github.com/frareb/myRBook_SP`).

Últimas modificaciones:

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*)

[1] `mailto:francois.rebaudo@ird.fr`

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)
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`). La compilación usa Travis CI (`https://travis-ci.org`). La versión en línea se aloja y actualiza a través de Netlify (`http://myrbooksp.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?

4.2 Este libro

4.3 Lectura adicional en español

- R para Principiantes, Emmanuel Paradis (`https://cran.r-project.org/doc/contrib/rdebuts_es.pdf`)
- xxx

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. La información adicional como esta nota siempre estará representada con este pictograma *información*.

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 *consola*.

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).

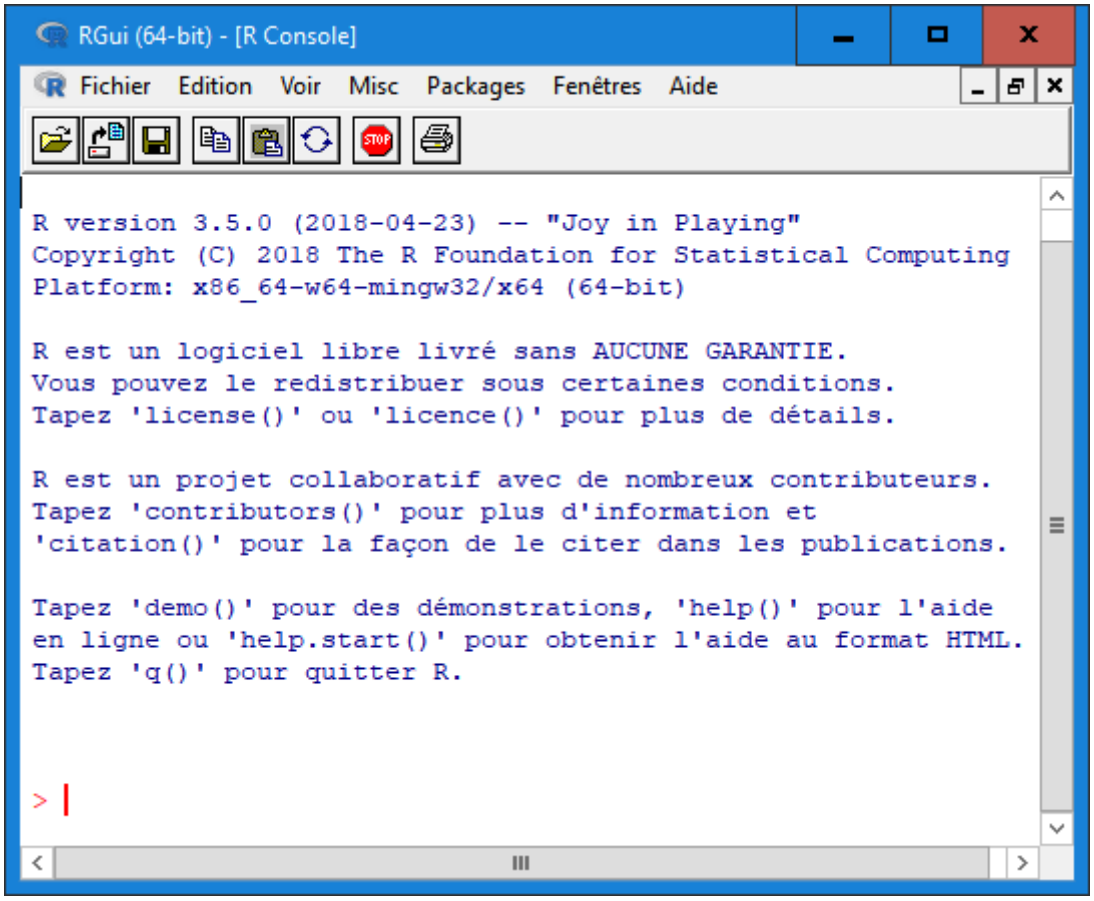

Figure 5.1: Captura de pantalla de la consola R en Windows.

Table 5.1: Operadores aritméticos.

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

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 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
```

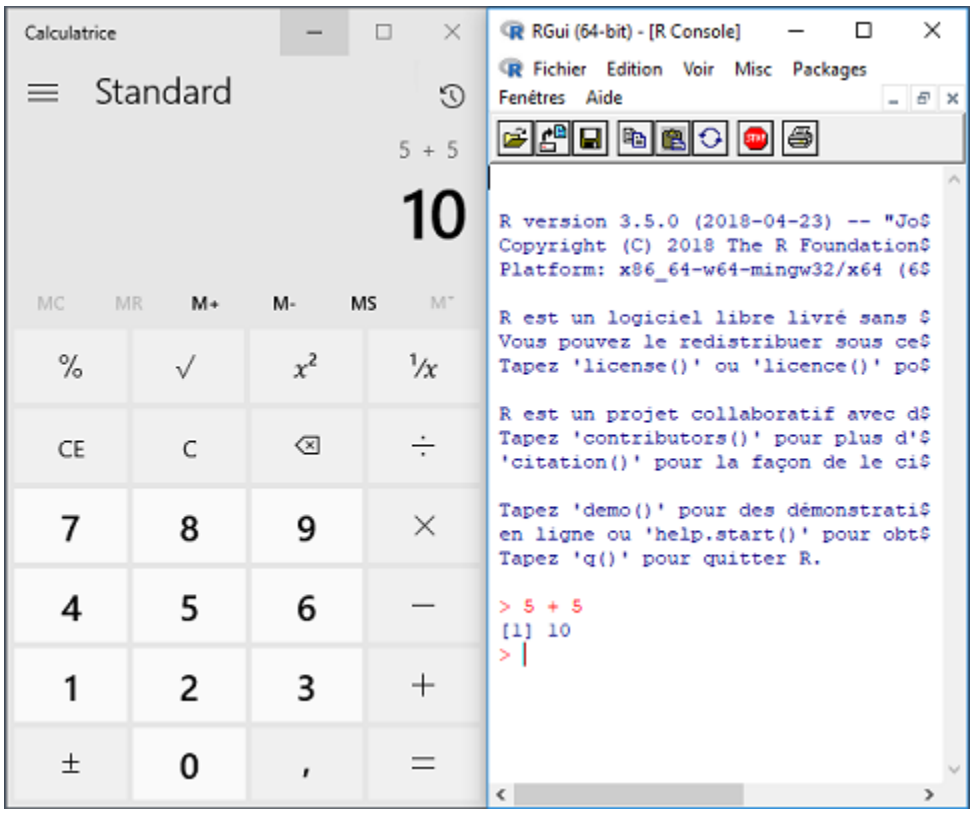

Figure 5.2: Captura de pantalla de la consola R al lado de la calculadora de Windows.

```
(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 nuestro ejemplos. La información sobre el *estilo* siempre estará representada con este pictograma para que sea fácilmente identificable.

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 numero 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
```

Table 5.3: Operadores lógicos.

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

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

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

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
```

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

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

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

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

```r
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/Objecto_(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`, asi 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 despues del simbolo numeral # no sera ejecutado. Significa que podriamos usar comentarios como ### o `#comentario`, aun que se recomienda hacer comentarios con un solo simbolo numeral seguido por un espacio y despues 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 secunda linea:

```
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 ingles ; `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. Sera es 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 utilisarlos 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.

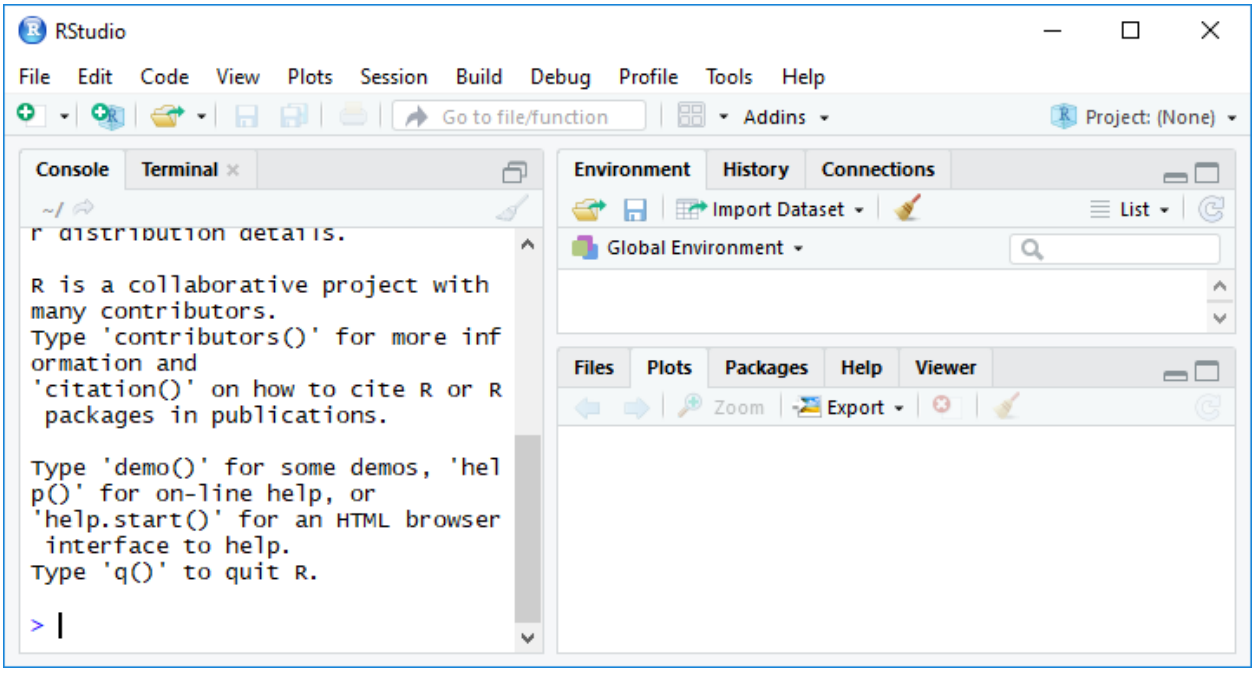

Figure 6.2: Captura de pantalla de RStudio en Windows: pantalla por defecto.

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 resaltado de preferencias de sintaxis predeterminado con RStudio. Estas preferencias de sintaxis pueden modificarse yendo al menú *Herramientas* (o *Tools*), *Opciones globales …* (o *Global Options…*), *Aspecto* (o *Appearance*), y luego seleccionando otro tema del *Editor de tema:* (o *Editor theme:*). Elegiremos el tema *Cobalt*, luego *OK* (Figura 6.4).

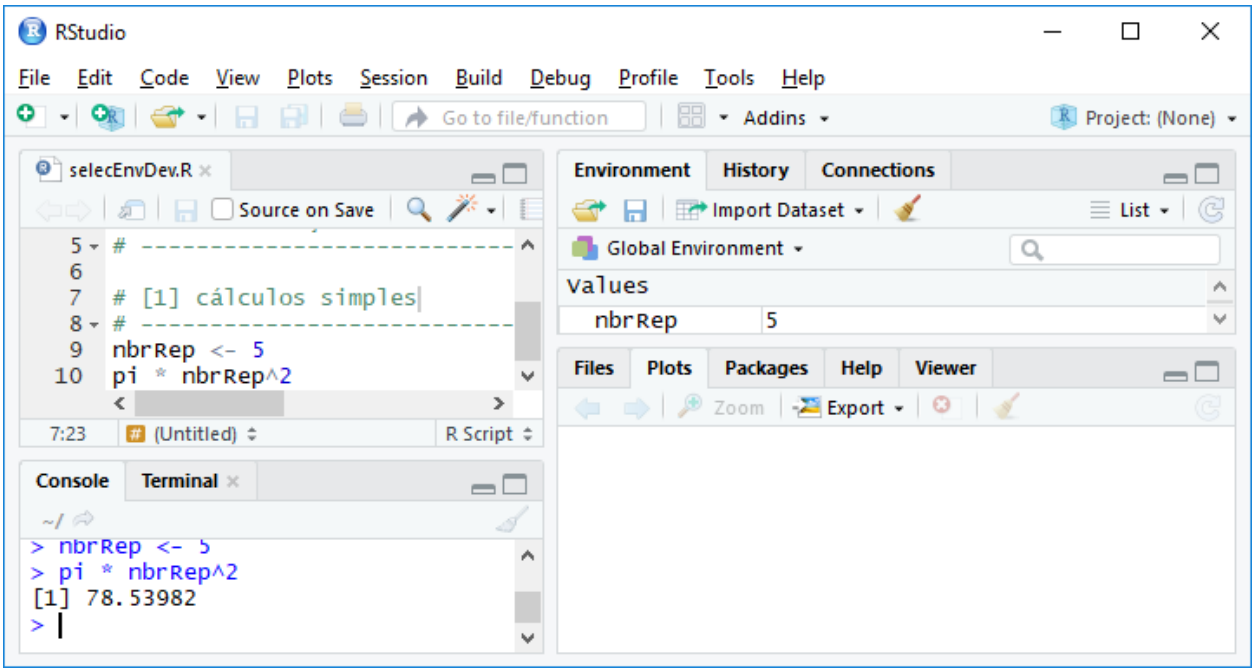

Figure 6.3: Captura de pantalla de RStudio en Windows: ejecutar nuestro script con CTRL + ENTER.

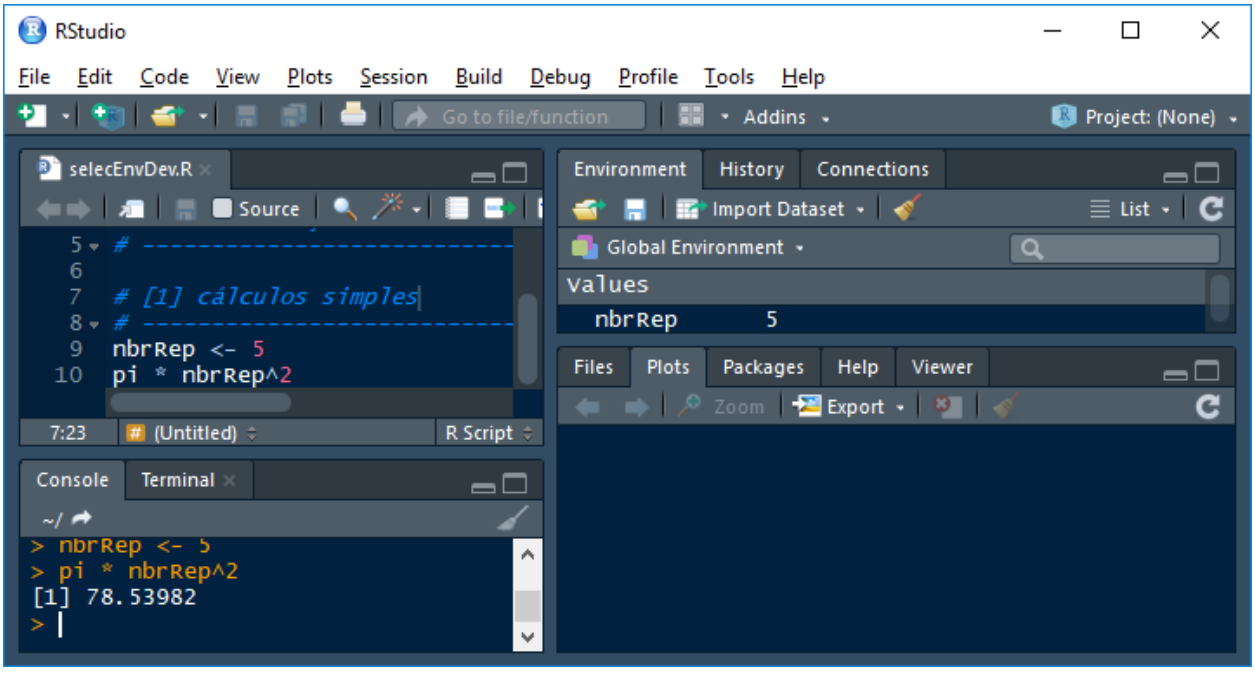

Figure 6.4: Captura de pantalla de RStudio en Windows: cambiar preferencias de sintaxis.

Figure 6.5: Logo Notepad++

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/npptor/`). Npp2R debe instalarse después de 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)

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.

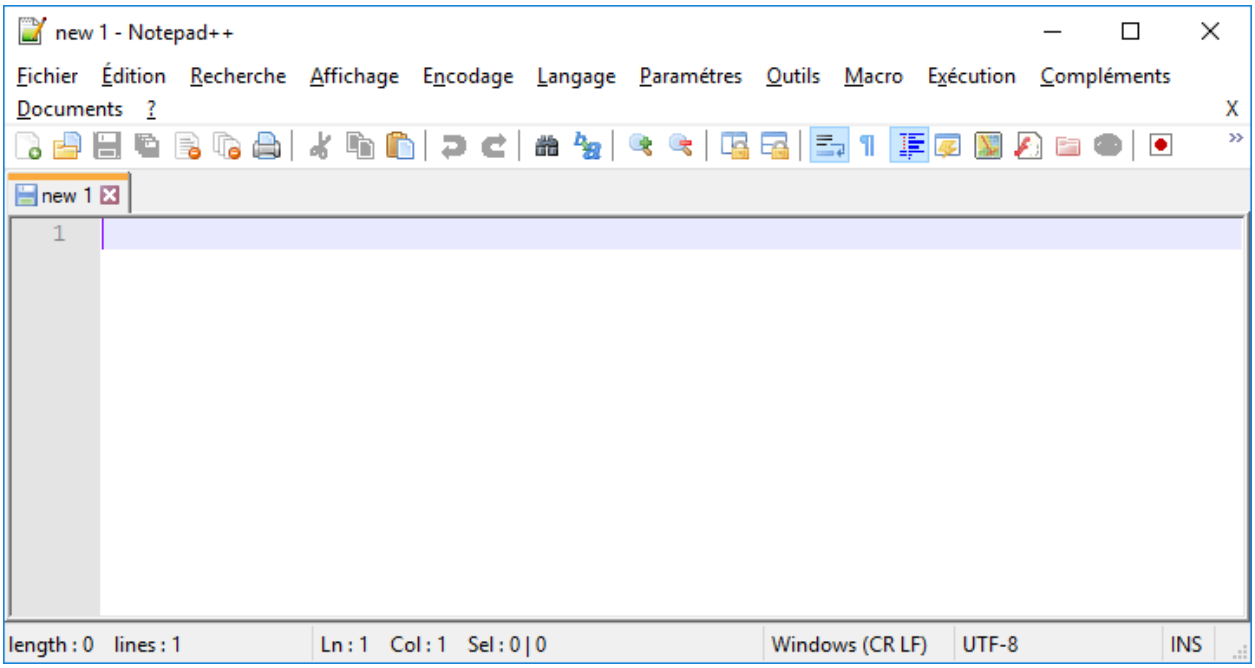

Figure 6.6: Captura de pantalla de Notepad++ en Windows: pantalla por defecto.

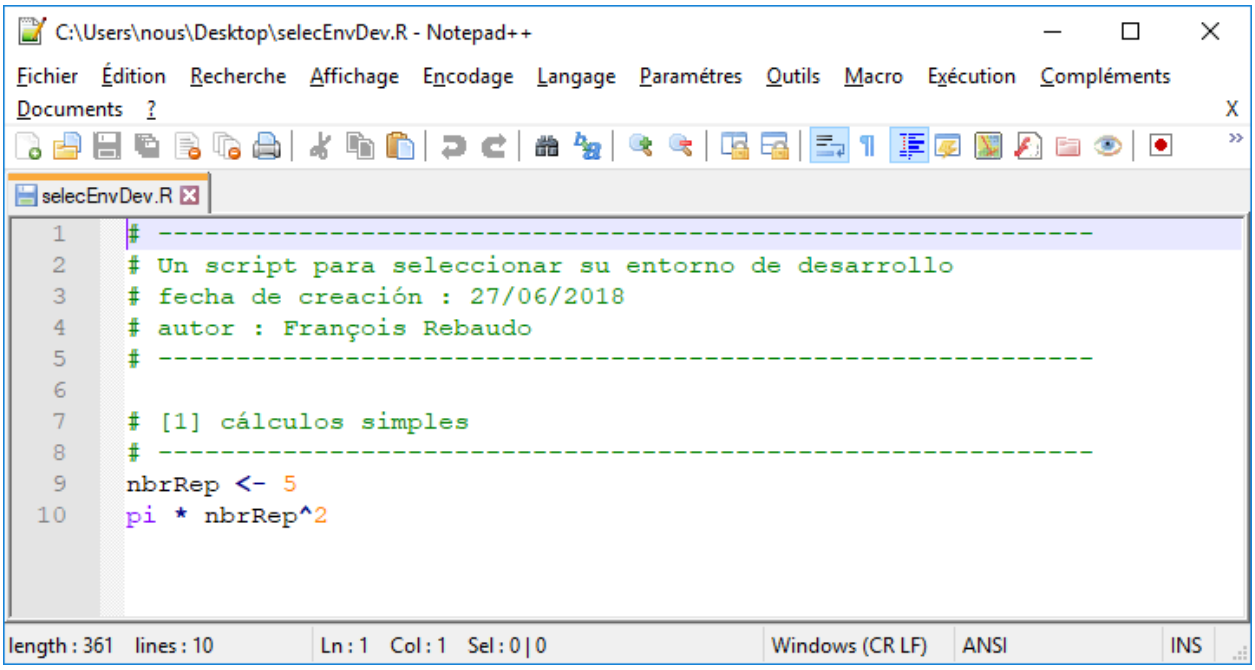

Figure 6.7: Captura de pantalla de Notepad++ en Windows: ejecutar nuestro script con F8.

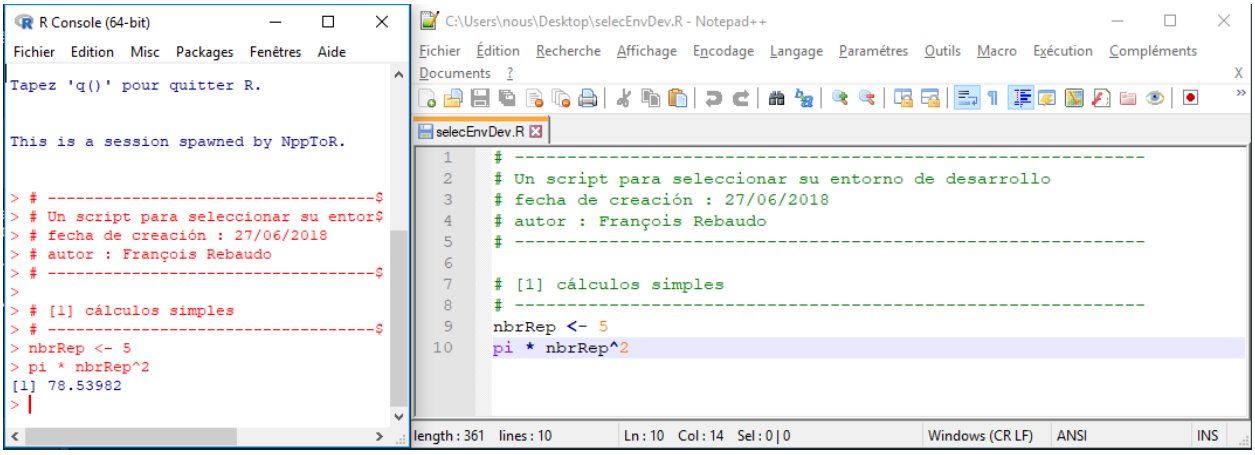

Figure 6.8: Captura de pantalla de Notepad++ en Windows: la consola con F8.

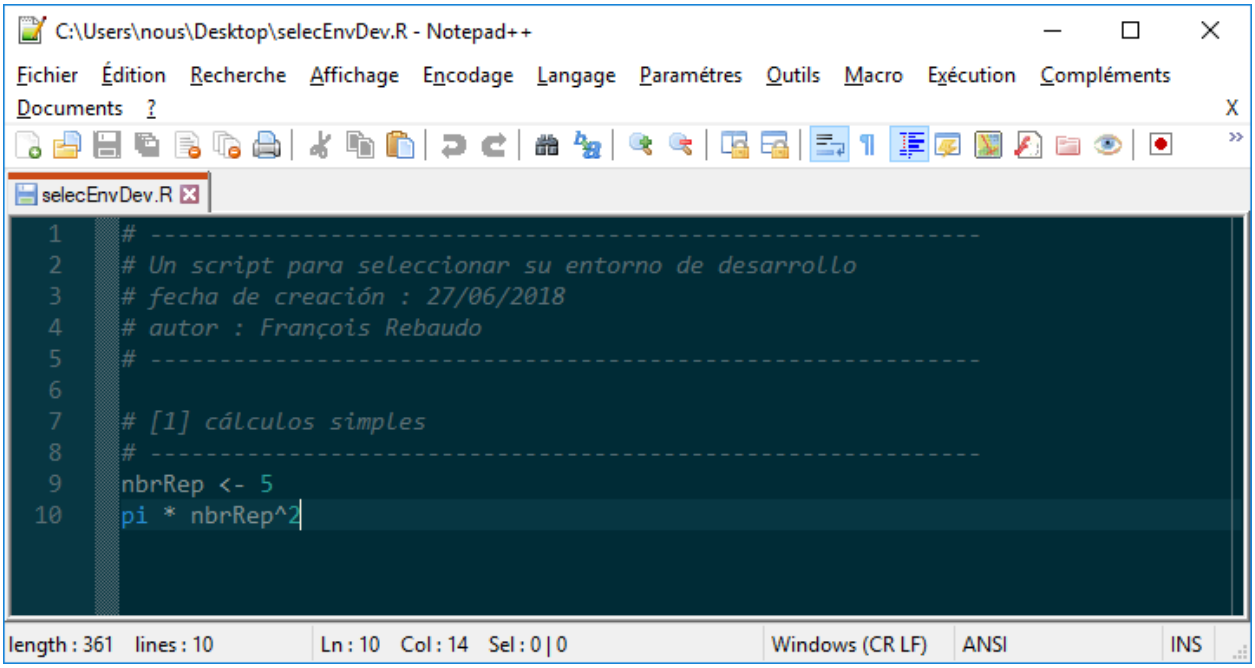

Figure 6.9: Captura de pantalla de Notepad++ en Windows con el tema Solarized.

Figure 6.10: Logo Geany

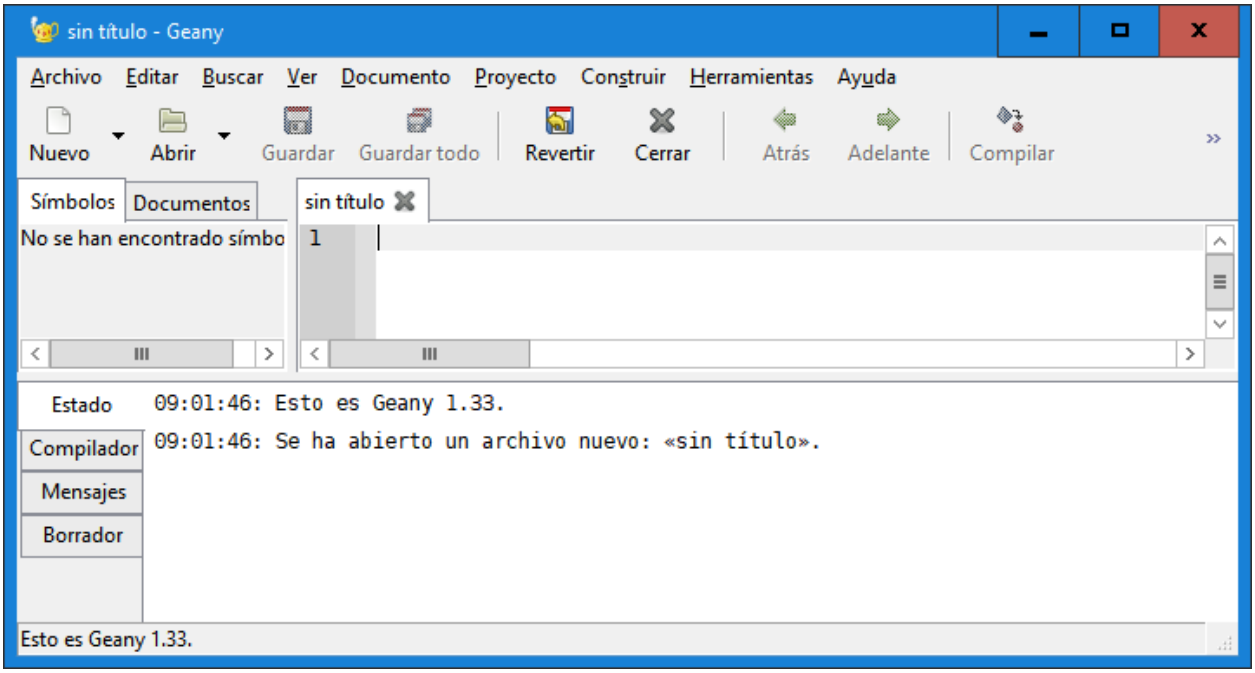

Figure 6.11: Captura de pantalla de Geany en Windows: pantalla por defecto.

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).

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:`

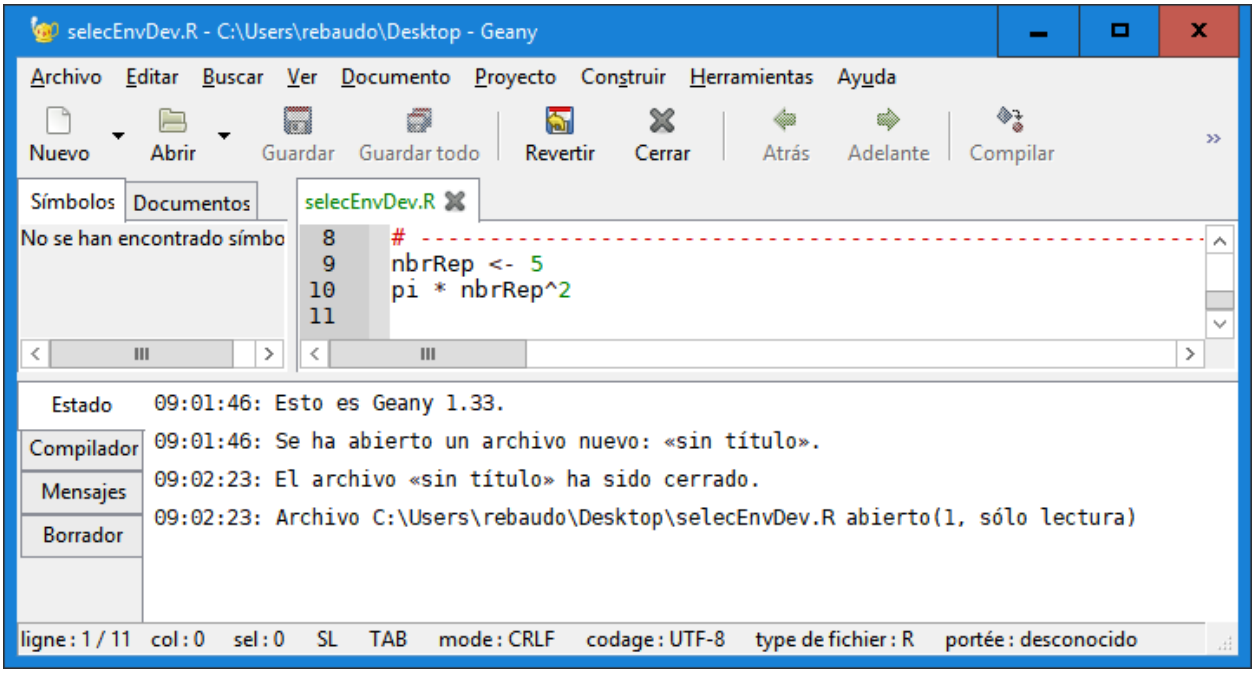

Figure 6.12: Captura de pantalla de Geany en Windows: abrir un script.

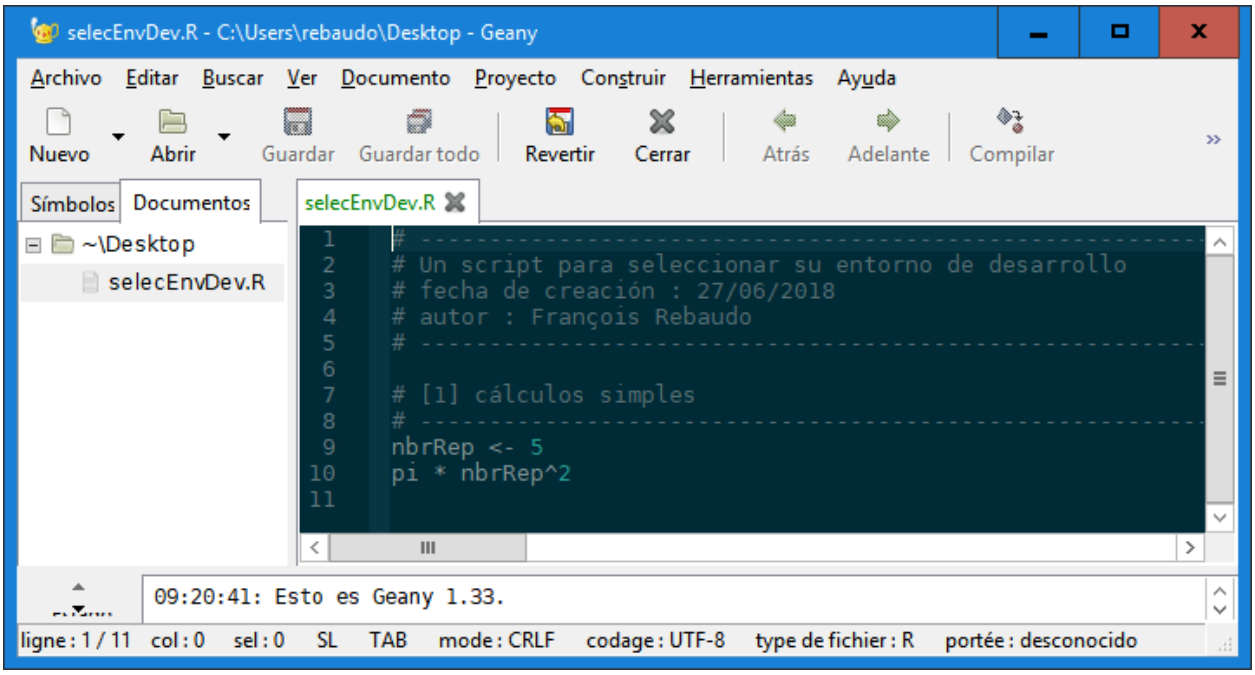

Figure 6.13: Captura de pantalla de Geany en Windows: cambiar esquema de color.

//www.sublimetext.com/), **Vim** (https://www.vim.org/), **Gedit** (https://wiki.gnome.org/Apps/Gedit), **GNU Emacs** (https://www.gnu.org/software/emacs/), 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

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.

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

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

```r
typeof(5.32)
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```r
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.

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 qsdfg hjklm'
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 qsdfg hjklm'
print(myText3)
```

```
## [1] "azerty uiop qsdfg hjklm"
```

```
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 funciónas`.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 funcion `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 selectionarlos. 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 corchetes 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
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"
```

```r
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`).

```r
# 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
```

```r
# 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 resaltará 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?

XXX

9.2 Las funciones más comunes

9.2.1 Ver los datos

str head tail names %in%

9.2.2 Manipular los datos

rank order sort

9.2.3 Funciones matemáticas

exp sqrt t

9.2.4 Estadísticas descriptivas

mean sd max min quartile summary median length round ceiling floor rowSums colSums rowMeans colMeans aggregate

9.3 Otras funciones útiles

seq_along : rep seq

9.4 Escribe una función

XXX argumentos

Chapter 10

Importar y exportar datos

10.1 Leer datos de un archivo

XXX

10.2 Guardar datos para R

save load

10.3 Exportar datos

write XXX

Chapter 11

Los bucles

11.1 ¿Por qué hacer bucles?

XXX algorítmico …

11.2 El bucle `if`

XXX

11.3 El bucle `switch`

XXX

11.4 El bucle `for`

XXX

11.5 El bucle `while`

XXX

11.6 `repeat`, `next`, `break`, `stop`

XXX

11.7 Los bucles de la familia `apply`

11.7.1 `apply`

XXX

11.7.2 `sapply`

XXX

11.7.3 `lapply`

XXX

11.7.4 `tapply`

XXX

11.7.5 `mapply`

XXX

Part II

Los gráficos

Chapter 12

Gráficos simples

12.1 plot

12.2 hist

12.3 barplot

12.4 boxplot

12.5 image y contour

Chapter 13

Gestión del color

13.1 colors()

13.2 RGB

13.3 Paletas

Chapter 14

Gráficos compuestos

14.1 mfrow

14.2 layout

Chapter 15

Manipular gráficos

15.1 Inkscape

15.2 The Gimp

Part III

Estadísticas con R

Chapter 16

Estadísticas descriptivas

Part IV

Estudio de caso

Chapter 17

Analizar datos de loggers de temperatura