Programación con R

Francisco Martínez del Río

2022-05-04

Contents

Chapter 1

Preámbulo

Este libro se ha escrito usando R Markdown y bookdown en el entorno RStudio. Un documento R Markdown es un documento de texto que permite combinar texto con formato y código en el lenguaje R (u otros lenguajes de programación) y los resultados que produce dicho código.

Chapter 2

Primeros pasos

En este primer tema vamos a ver algunas utilidades generales del entorno R usando RStudio, tales como la ejecución de órdenes en la consola, la obtención de ayuda y la instalación y uso de paquetes.

2.1 Ejecución de órdenes en la consola

Para ejecutar órdenes en la consola teclea la expresión deseada y pulsa Enter.

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

Observa que la salida viene precedida por [1]. Para esta salida este valor no es muy útil, pero sí lo es cuando la salida produce un resultado que abarca varias líneas. Por ejemplo:

```
(1:40) ^ 2
##  [1]    1    4    9   16   25   36   49   64   81  100  121  144  169  196  225
## [16]  256  289  324  361  400  441  484  529  576  625  676  729  784  841  900
## [31]  961 1024 1089 1156 1225 1296 1369 1444 1521 1600
```

Esto produce los números del 1 al 40 elevados al cuadrado. Cada línea viene precedida por el ordinal del primer valor de la línea en el conjunto de datos de salida. Por ejemplo, si una línea comienza con [14] eso indica que el primer valor en esa línea ocupa el puesto 14 de los valores que aparecen en la salida.

Cuando se escribe en la consola, se puede utilizar las flechas hacia arriba y abajo para recuperar las distintas órdenes tecleadas. Pruébalo, es realmente útil. También es cómodo usar la pestaña **History** del panel superior derecho para

visualizar las órdenes tecleadas previamente. Si pinchamos en una orden aparece resaltada y se puede usar las opciones **To Console** o **To Source** para enviar la orden a la consola o al archivo de texto que estemos editando respectivamente. Pruébalo, a veces evita el tener que volver a teclear una orden larga escrita con anterioridad.

Observa que en la consola aparece el símbolo >. Este símbolo es el indicador o *prompt*. Indica que el sistema está listo para recibir órdenes. Teclea una orden incompleta como `7 *` y pulsa *Enter*. El indicador cambia al símbolo +. Este símbolo nos indica que la línea previa era incompleta y está esperando el final de la entrada. Escribe `5`, por ejemplo, y pulsa *Enter*, se obtendrá el valor `35`. Si cuando aparece el indicador + deseas abortar la orden que estabas escribiendo, en lugar de completarla, pulsa *Escape*.

2.2 Ayuda

Uno de los aspectos sobresalientes de R es su sistema de ayuda. Se puede pedir ayuda sobre cualquier función, paquete, instrucción, conjunto de datos, etc. Para ello hay que usar la función `help`:

```
help("mean") # ayuda sobre la función mean
```

Si estamos en RStudio, la ayuda aparece en la pestaña **Help** del panel inferior derecho. Para obtener ayuda, también se puede usar la interrogación:

```
?var
```

La ayuda sobre funciones incluye descripciones de sus parámetros de entrada y valor de retorno. La última parte de la ayuda (*examples*) es muy interesante, pues incluye ejemplos de uso de la función. Estos ejemplos se pueden ejecutar en la consola con la función `example`:

```
example(seq)
```

A lo largo de estos apuntes vamos a estudiar muchas funciones de R. Normalmente describiremos los usos más comunes de una función. Sin embargo, hay funciones muy flexibles que realizan cálculos muy variados en función de cómo se las invoque. Por ello te animamos a que uses la ayuda para obtener una descripción completa de las funciones que te interesen. Prueba, por ejemplo, `help("help")` para obtener una descripción más detallada del funcionamiento de la propia función `help`.

Aparte de esta ayuda, y de la que se explica a continuación, una búsqueda en internet usando unos términos significativos, preferiblemente en inglés, suele producir ayuda de gran calidad en lugares como *stackoverflow*.

2.2.1 Vignettes

Los paquetes de R están obligados a incluir documentación sobre sus funciones y conjuntos de datos. Sin embargo, a veces es difícil entender la utilidad de un paquete viendo la documentación de sus funciones de forma aislada. Es por ello que algunos paquetes incluyen *vignettes*, que son documentos que explican en detalle la funcionalidad de un paquete combinando sus distintas funciones para, por ejemplo, resolver un problema. Un paquete puede contener varias *vignettes*. Para consultarlas desde la consola de R se puede usar la función `vignette`.

```
vignette("grid")
```

Si ejecutas la orden previa desde la consola, se abrirá un archivo pdf con información sobre el sistema gráfico *grid*.

2.2.2 Demos

La tercera utilidad que nos ofrece R para aprender a usar sus recursos son las demostraciones, es decir, ejemplos prácticos en los que se muestran los resultados de utilizar ciertas funciones. Muchos paquetes incorporan demostraciones bastante elaboradas sobre su funcionalidad, especialmente los paquetes que sirven para generar gráficos.

Como ejemplo, ejecuta la siguiente orden en la consola y verás una demostración de las posibilidades gráficas de R (los gráficos aparecerán en la pestaña **Plots** del panel inferior derecho):

```
demo(graphics)
```

2.3 Paquetes

El núcleo base de R viene constituido por una serie de paquetes que se incluyen en la instalación de R. Este núcleo base incorpora un gran abanico de funcionalidades con las que podemos cargar datos de fuentes externas, llevar a cabo análisis estadísticos y obtener representaciones gráficas. Las funcionalidades de los paquetes del núcleo base están disponibles al ejecutar RStudio. No obstante, hay multitud de tareas para las que necesitaremos recurrir a paquetes externos al núcleo base, y necesitaremos incorporar al entorno de trabajo las funciones y objetos definidos en ellos. Algunos de esos paquetes ya se encontrarán instalados en el sistema, pero otros será preciso descargarlos e instalarlos conforme se vayan necesitando. Una de las grandes ventajas de R es la gran comunidad de programadores que desarrolla paquetes para compartirlos con el resto. Además, comparado con otros sistemas, la instalación de paquetes es muy sencilla.

Un paquete es un conjunto de funciones y datos listos para ser distribuidos e instalados. De cara a instalar un paquete hay varias posibilidades. Una de ellas es usar la pestaña **Packages** del panel inferior derecho. Mediante esta pestaña podemos consultar los paquetes que tenemos instalados, ver su documentación e instalar nuevos paquetes. Otra posibilidad es usar la función `install_packages`. Por ejemplo, la siguiente orden instala el paquete *tsfknn*.

```
install.packages("tsfknn")
```

La función `install_packages` descarga de Internet e instala un paquete de CRAN, que es el repositorio oficial de paquetes de R.

2.3.1 Carga de paquetes

Para poder usar un objeto (función, constante, conjunto de datos, ...) de un paquete es necesario tener instalado el paquete; pero no sólo eso, también hay que cargar el paquete, para que los objetos que contiene sean accesibles. Podemos consultar los paquetes cargados con la función `search`.

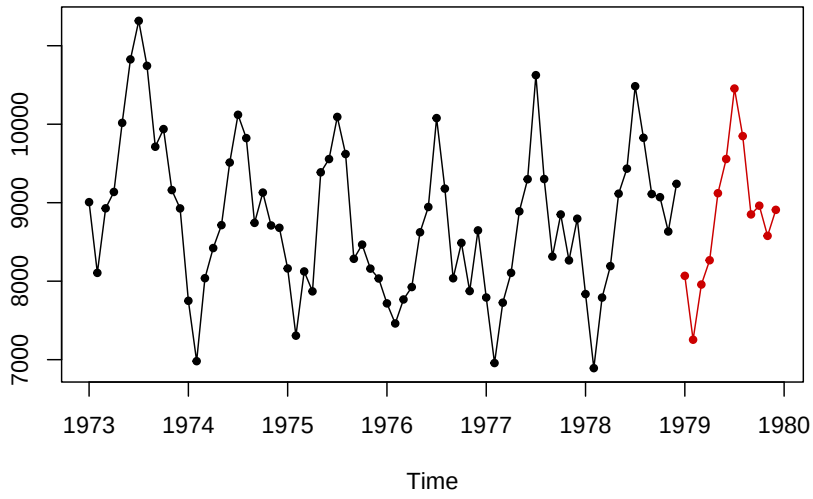
```
search()
## [1] ".GlobalEnv"        "package:stats"     "package:graphics"
## [4] "package:grDevices" "package:utils"     "package:datasets"
## [7] "package:methods"   "Autoloads"         "package:base"
```

Para cargar un paquete y tener acceso a sus objetos hay que usar la función `library`. Por ejemplo:

```
library(tsfknn)
```

Ahora podemos usar las funciones del paquete *tsfknn*, que, dada una serie temporal, predice su comportamiento futuro.

```
pred <- knn_forecasting(USAccDeaths, h = 12, lags = 1:12)
plot(pred)
```



Podemos comprobar que el paquete se ha cargado usando de nuevo la función `search`:

```
search()
##  [1] ".GlobalEnv"        "package:tsfknn"    "package:stats"
##  [4] "package:graphics"  "package:grDevices" "package:utils"
##  [7] "package:datasets"  "package:methods"   "Autoloads"
## [10] "package:base"
```

Ahora el paquete *tsfknn* aparece en segundo lugar.

Como puede observarse, hay una serie de paquetes, como *graphics*, *utils*, *stats* o *base*, que no hemos cargado con `library` y, sin embargo, están cargados. Estos se corresponden con los paquetes del núcleo base de R y se cargan automáticamente. Estos paquetes incluyen funciones básicas para realizar cálculos y análisis estadísticos.

2.3.2 Conflictos de nombres

A veces más de un paquete puede contener un objeto con el mismo nombre. Esto provoca un conflicto de nombres porque, cuando usamos un nombre definido en más de un paquete, ¿a qué objeto nos referimos?. R lo resuelve usando el objeto del último paquete que ha sido cargado, que será el primero de los paquetes en conflicto que aparezca en el listado producido por la función `search`.

Vamos a ejemplificar un conflicto de nombres sin necesidad de usar dos paquetes. La función `mean`, del paquete `base` del núcleo de R, calcula la media de los valores de un vector:

```
mean(c(2, 7, 10))  # media del vector [2, 7, 10]
## [1] 6.333333
```

Sin embargo, a continuación vamos a escribir una función llamada `mean` que siempre produce el valor 5.

```
mean <- function(x) 5
```

Ahora, esta nueva función oculta a la función original del paquete `base`:

```
mean(c(2, 7, 10))
## [1] 5
```

Cuando haya conflictos de nombres, siempre podemos usar la siguiente sintaxis para evitar el conflicto: `nombrePaquete::objeto`. Es decir, precedemos el nombre del objeto con el nombre del paquete en el que se ubica y dos ocurrencias del carácter :. Esto evita cualquier conflicto, pues no puede haber dos paquetes con el mismo nombre.

```
mean(c(2, 7, 10))        # invoca a la función que hemos definido
## [1] 5
base::mean(c(2, 7, 10))  # invoca a la función mean del paquete base
## [1] 6.333333
```

La sintaxis `nombrePaquete::objeto` también es útil para usar un objeto de un paquete sin necesidad de cargar previamente el paquete con `library`.

2.4 Ejercicios

1. Usa la función `help` para descubrir la utilidad de la función `sqrt`. Usa `sqrt` para realizar algún cálculo.
2. Genera una *vignette* del paquete *tsfknn*.
3. Instala el paquete `dlstats`, que sirve para consultar estadísticas sobre las veces que los usuarios instalan determinados paquetes en RStudio. Carga el paquete y obtén ayuda de la función `cran_stats`. Utiliza la función `cran_stats`para consultar un histórico de cuántas veces al mes se ha descargado el paquete *tsfknn*.

4. Ejecuta el código autoexplicativo de más abajo. El efecto es que la nueva variable oculta a la constante `pi`, definida en el paquete `base`. Añade una línea al final del código de forma que se pueda volver a consultar el valor de la constante π.

```
pi       # consulta el valor de la constante pi
## [1] 3.141593
pi <- 0  # variable que pone pi a cero (oculta a la otra)
pi       # consulta la variable que vale 0
## [1] 0
```

2.5 Soluciones

```
# Función sqrt
help("sqrt")
sqrt(25)
```

```
# Vignette del paquete tsfknn
vignette("tsfknn")
```

```
# Instalación y uso del paquete dlstats
install.packages("dlstats")  # se instala
library(dlstats)             # se carga
help("cran_stats")
cran_stats("tsfknn")
```

```
# Ocultación del objeto pi del paquete base
pi
## [1] 0
pi <- 0
pi
## [1] 0
base::pi
## [1] 3.141593
```

Chapter 3

Variables y expresiones

En este tema se estudia el concepto de variable. Las variables hacen referencia a datos. Los datos se pueden generar usando expresiones, que es el otro aspecto que se analiza en este tema.

3.1 Variables

Una variable permite asociar un valor u objeto a un identificador. En la memoria del ordenador habrá una zona donde se almacena el objeto asociado a la variable.

Para crear una variable en R hay que asignarle un valor mediante una operación llamada asignación. La asignación consta de tres elementos:

- el identificador o nombre de la variable
- el operador de asignación
- una expresión que produce el valor a asignar

Vamos a ver un ejemplo:

```
x <- 7
```

Aquí la variable tiene el nombre `x`, el operador de asignación está compuesto por los caracteres < y - escritos juntos (`<-`). Por último, la expresión es el valor 7. A partir de ahora, la variable `x` tiene asociado el valor 7 y podemos usar la variable para consultar su valor. Por ejemplo:

```
y <- x * 6
```

Ahora, la variable `x` aparece en una expresión. Al evaluar la expresión la variable `x` se sustituye por su valor (7), de forma que `x * 6` produce el valor 42, que es asignado a la variable `y`.

Como escribir los caracteres < y - resulta tedioso, RStudio tiene un atajo (una combinación de teclas) para teclearlo más rápidamente. Consiste en pulsar la tecla *Alt* junto a -.

Si tecleas una variable en la consola y pulsas *Enter*, R evalúa la expresión formada por la variable, lo cual equivale a consultar su valor.

```
y
## [1] 42
```

En la pestaña **Environment**, del panel superior derecho, aparecen por defecto los objetos creados en el Entorno Global o espacio de trabajo, que incluyen las variables que se han creado en la sesión de trabajo. Ahí puedes consultar las variables creadas y su valor.

3.1.1 Nombres de variables

El nombre de una variable es un identificador. Un identificador también puede ser, por ejemplo, el nombre de una función. Las siguientes reglas se aplican a los identificadores:

- Pueden ser una combinación de letras, dígitos, punto (.) y subrayado (_).
- Deben comenzar por una letra o punto. Si comienzan con punto, el siguiente carácter no puede ser un dígito.
- Las palabras reservadas de R, como `if` o `for` no pueden usarse como identificadores. (Ejecuta `?Reserved` para consultar las palabras reservadas).

Ejemplos de identificadores válidos son: `total`, `Suma`, `.bien`, `valor_total`,`nombre5`. Ejemplos de identificadores inválidos son: `alumn@`, `7nombre`, `_bien`, `TRUE`, `.6nombre` (`TRUE` no es válido porque es una palabra reservada de R).

R es sensible a las mayúsculas. Esto significa que los identificadores `Nombre`, `nombre` o `NOMBRE` son distintos. Aunque R lo permite, no uses identificadores que sólo se diferencian en si unas letras están en mayúsculas o minúsculas, pues puede causar confusión.

3.1.2 Asignando distintos valores a una variable

A una variable se le puede asignar distintos valores a lo largo del tiempo. Sin embargo, la variable sólo guarda el último valor asignado. Veamos un ejemplo (en el ejemplo las asignaciones se encierran entre paréntesis para que el resultado de la asignación aparezca en la pantalla):

```
(z <- 20)
## [1] 20
(z <- 6)
## [1] 6
z
## [1] 6
```

R es un lenguaje con tipado dinámico. Esto significa que una variable puede contener valores de distinto tipo a lo largo del tiempo (estudiaremos los distintos tipos de datos más adelante). Por ejemplo:

```
a <- 4.5     # a contiene un número
a <- "Juan"  # después una cadena de caracteres
a <- TRUE    # y ahora un valor lógico
```

3.1.3 Operadores de asignación alternativos

Aparte del operador de asignación: `<-`, existen dos operadores más: `=`y `->`:

```
b = 9
16 -> c
```

Como puede observase, cuando se usa el operador `->` la expresión aparece a la izquierda del operador y la variable a la derecha. De momento, se recomienda usar el operador `<-`.

3.1.4 Consulta y borrado de variables

Para consultar las variables y, en general, los objetos que hemos definido, podemos usar la función `ls` y `objects`:

```
ls()
## [1] "a"    "b"    "c"    "mean" "pi"   "pred" "x"    "y"    "z"
```

También podemos consultar los objetos definidos en la pestaña **Environment** del panel superior derecho de RStudio. Para borrar variables del espacio de trabajo se puede usar la función `rm`. Por ejemplo, vamos a borrar las variables `a` y `b`:

```
rm(a, b)
ls()
## [1] "c"    "mean" "pi"   "pred" "x"    "y"    "z"
```

Si queremos borrar todos los objetos que hemos definido podemos usar la escoba que aparece en la pestaña **Environment** del panel superior derecho. Otra posibilidad es esta:

```
rm(list = ls())
ls()
## character(0)
```

3.1.5 Guardar y recuperar objetos del entorno de trabajo

Normalmente, cuando terminamos una sesión de trabajo en R no estamos interesados en guardar los distintos objetos que hemos creado. Sin embargo, hay veces que algunas variables almacenan resultados que queremos mantener, porque para obtenerlos se precisa mucho trabajo y/o tiempo. En esos casos podemos recurrir a la función `save`:

```
a <- -5.5
b <- 22
c <- 0
save(a, b, file = "datos.RData")
```

La función `save` guarda una serie de variables en un archivo en memoria secundaria (cuyo contenido no se borra al apagar el ordenador). Como se puede ver, se listan los nombres de las variables que queremos guardar (hemos elegido guardar las variables `a` y `b`) y el nombre del archivo donde se guardarán los datos. Se suele usar la extensión RData para estos archivos. Para recuperar los datos almacenados con `save` se usa la función `load`:

```
rm(list = ls())  # borramos todas las variables
load("datos.RData")
ls()
## [1] "a" "b"
```

Hemos recuperado las variables `a` y `b`. Otra forma de cargar un archivo de datos es localizarlo en la pestaña **Files** en el panel inferior derecho y hacer doble clic en él o usar la opción **Session>Load Workspace ...**.

Si se quiere guardar todo el espacio de trabajo, se puede usar la función `save.image`:

```
rm(list = ls())  # borramos todas las variables
x <- 9
y <- 28
save.image(file = "todas.RData")
```

`save.image` usa por defecto el nombre de archivo `.RData`. Como curiosidad, cuando cerramos RStudio se nos preguntan si ejecutar `save.image` con sus parámetros por defecto. Si decimos que sí, automáticamente se guardan nuestros objetos en un archivo llamado `.RData` en el directorio de trabajo. Cuando se abre RStudio se recuperan los datos almacenados en un archivo `.RData` situado en la carpeta de inicio de RStudio.

```
rm(list = ls())  # borramos todas las variables
load("todas.RData")
ls()
## [1] "x" "y"
```

Otra posibilidad para guardar todo el espacio de trabajo es usar el botón con un disco del panel **Environment**. Las funciones `savehistory` y `loadhistory` también nos permiten guardar y recuperar el historial de órdenes tecleadas en la consola.

Cuando leemos datos de un archivo con `load` se pueden cargar varias variables. Podemos consultar los nombres de dichas variables así:

```
nombres <- load("todas.RData")
nombres
## [1] "x"             "y"             ".Random.seed"
```

Si sólo queremos guardar un objeto en un archivo, se puede usar la función `saveRDS`. Una pequeña ventaja de usar esta función es que al cargar el objeto (con `readRDS`) podemos elegir su nombre:

```
n <- "Julio"
saveRDS(n, file = "objeto.RDS")
nombre <- readRDS("objeto.RDS")
nombre
## [1] "Julio"
```

3.2 Expresiones aritméticas

Las variables son útiles pues permiten almacenar valores y resultados de cálculos, resultados que podemos consultar o usar con posterioridad. Las expresiones sirve para expresar cálculos. Las vamos a usar continuamente, pues el objeto de usar el ordenador suele ser realizar cálculos. A menudo, el resultado de evaluar expresiones se almacena en variables.

Las expresiones aritméticas expresan cálculos numéricos. Como hemos visto, pueden usarse en asignaciones. El resultado de evaluar la expresión es lo que se asigna a la variable. Una expresión aritmética puede contener:

- valores literales: son valores concretos, como el valor 7.
- variables: al evaluar la expresión una variable se sustituye por su valor asociado.
- operadores aritméticos como: +, -, *, /, ^
- llamadas a funciones: las funciones producen valores en función de sus parámetros.

Veamos algunos ejemplos de expresiones aritméticas:

```
20            # literal
## [1] 20
1.8 * 2       # literales y operador de multiplicación
## [1] 3.6
x <- 9
x ^ 3         # variable, operador y literal
## [1] 729
exp(x) - 1000 # función exponencial, operador y literal
## [1] 7103.084
```

3.2.1 Valores literales numéricos

Un valor literal numérico es un número concreto, también a veces llamado escalar. Existen varias formas de escribir un literal de tipo real:

- En decimal, por ejemplo: `0.18`.
- En notación científica, por ejemplo: `1.23e4`. Este número equivale a $1.23 * 10^4$, o sea, 12300.
- En hexadecimal, por ejemplo: 0x2f. Este número es el 47 en decimal $(2 * 16^1 + 15 * 16^0)$.

Para escribir un literal entero se procede como con los reales, pero no se puede usar parte fraccional y hay que terminar el número con `L`. Por ejemplo: `12L`, `123e2L`, `0x2fL`.

3.2.2 Precedencia y asociatividad de los operadores

Las expresiones de ejemplo previas incluyen un único operador. Sin embargo, una expresión puede incluir más de un operador. Las reglas de precedencia indican el orden en que se aplican los operadores. Por ejemplo, la multiplicación (operador *) tiene mayor precedencia que la suma (operador +):

```
2 + 3 * 5
## [1] 17
```

Como la multiplicación tiene mayor precedencia se ha calculado primero `3 * 5` (15) y despues `2 + 15`. Como en matemáticas, los paréntesis sirven para alterar el orden de precedencia:

```
(2 + 3) * 5
## [1] 25
```

La asociatividad de un operador se usa cuando aparece más de una vez seguida un operador, y sirve para determinar si el operador se aplica de izquierda a derecha o de derecha a izquierda. Por ejemplo, el operador de división (/) se asocia de izquierda a derecha:

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

Como la asociatividad es de izquierda a derecha, primero se calula `8 / 2` (4) y después `4 / 4` (1). De nuevo, los paréntesis sirven para alterar el orden de aplicación de los operadores:

```
8 / (2 / 4)
## [1] 16
```

A continuación se listan los distintos operadores aritméticos y lógicos de R ordenados de mayor a menor precedencia, en la última columna se indica su asociatividad.

Operador	Significado	Asociatividad
^	Potencia	Derecha a izquierda
-x, +x	menos unario, más unario	Izquierda a derecha
%%	Módulo	Izquierda a derecha
*, /	Multiplicación, división	Izquierda a derecha
+, –	Suma, resta	Izquierda a derecha
<, >, <=, >=, ==, !=	Relacionales	Izquierda a derecha
!	NO lógico	Izquierda a derecha
&, &&	Y lógico	Izquierda a derecha
|, ||	O lógico	Izquierda a derecha
->, -»	Asignación a la derecha	Izquierda a derecha
<-, «-	Asignación a la izquierda	Derecha a izquierda
=	Asignación a la izquierda	Derecha a izquierda

Todos los operadores de la lista son binarios, es decir, tiene dos operandos, salvo los operadores + y -. Estos tienen una versión binaria (para sumar y restar) y una versión unaria (para indicar el signo). Por ejemplo:

```
-2 + 5
## [1] 3
```

En esta expresión el - es el operador unario y el + es el operador binario. Observa que el resultado es 3, porque el operador unario tiene más precedencia que el binario. Si la precedencia fuera al revés el resultado sería -7. Otro ejemplo sería:

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

En este caso el primer - es el operador binario y el segundo el unario.

Teniendo en cuenta la precedencia y asociatividad. ¿Qué crees que produce `-2 ^ 2` (4 ó -4)?, ¿Y `2 ^ 3 ^ 3` (512 ó 134217728)? Prueba en la consola si has acertado. Usa paréntesis para obtener el otro resultado.

3.3 Expresiones lógicas o booleanas

En programación se usa con gran asiduidad las operaciones lógicas, también conocidas como booleanas. El término booleano viene del nombre del matemático George Boole. Una operación lógica es aquella que produce un valor de verdadero o falso. Existen dos literales booleanos, que son los valores `TRUE` y `FALSE`.

```
(casado <- TRUE)
## [1] TRUE
(americano <- FALSE)
## [1] FALSE
```

Una variable como `casado` o `americano`, que almacena un valor lógico, es llamada variable lógica o booleana. Se puede abreviar los valores `TRUE` y `FALSE` con `T` y `F` respectivamente, pero resulta más legible usar los primeros. Además, `T` y `F` son realmente variables que almacenan los valores lógicos `TRUE` y `FALSE` respectivamente. Al ser variables, sus valores pueden ser sobrescritos. No ocurre lo mismo con `TRUE` y `FALSE` que son palabras reservadas y no pueden ser sobrescritas.

3.3.1 Operadores relacionales

La mayor parte de las expresiones lógicas incluyen operadores relacionales. Los operadores relacionales son binarios y sirven para comparar dos valores. Los operadores relacionales son:

- <: menor
- <=: menor o igual
- >: mayor
- >=: mayor o igual
- ==: igual (observa que son dos símbolos de =)
- !=: distinto

Veamos un ejemplo de uso de algunos de estos operadores:

```
x <- 10
x > 0      # ¿es x mayor que 0?
## [1] TRUE
x != 10    # ¿es x distinto de 10?
## [1] FALSE
```

3.3.2 Operaciones lógicas compuestas

Los operadores lógicos permiten generar expresiones lógicas compuestas a partir de expresiones lógicas simples. Los principales operadores lógicos son el operador Y, el O y el NO.

3.3.2.1 Operador Y (and): `&&`

Es un operador binario. Produce el valor verdadero si ambos operandos son verdaderos, en otro caso produce el valor falso. Ejemplo:

```
x <- 10
y <- -5
x > 0 && y < 0    # ¿es x mayor que cero e y menor que cero?
## [1] TRUE
x == 10 && y == 0 # ¿es x igual a 10 e y igual a 0?
## [1] FALSE
```

El operador es &&. El operador & es equivalente, pero se usa para colecciones de datos, como vectores.

3.3.2.2 Operador O (or): ||

Es un operador binario. Produce un valor verdadero si al menos uno de los operandos es verdadero, en otro caso (ambos operandos son falsos) produce el valor falso. Ejemplo:

```
x <- 10
y <- -5
x > 0 || y > 0
## [1] TRUE
x == 0 || y == 0
## [1] FALSE
x > 0 || y < 0
## [1] TRUE
```

El operador es ||. El operador | es equivalente, pero se usa para colecciones de datos, como vectores.

3.3.2.3 Operador NO (not)

Es un operador unario, es decir, tiene un único operando. Invierte el valor de su operando. Es decir, si el operando vale `TRUE` produce `FALSE`, y si el operando vale `FALSE` produce `TRUE`. Ejemplo:

```
casado <- TRUE
! casado
## [1] FALSE
```

El operador de negación tiene el símbolo !.

3.3.3 Ejemplos de expresiones lógicas

Escribir la expresión lógica oportuna para expresar si se verifica cierta condición lógica cuesta al principio. A continuación vamos a estudiar algunos ejemplos. Antes de ver la solución intenta pensar la expresión lógica adecuada, después consulta el código con la solución. Ten en cuenta que una condición lógica se puede expresar con distintas expresiones lógicas.

3.3.3.1 ¿Pertenece una variable a un intervalo?

Se trata de ver si el valor almacenado una variable está comprendido en un intervalo, por ejemplo, $[5, 10]$:

```
# Solución
x <- 7
x >= 5 && x <= 10 # ¿pertenece x al intervalo [5, 10]
## [1] TRUE
x <- 12
x >= 5 && x <= 10 # ¿pertenece x al intervalo [5, 10]
## [1] FALSE
```

3.3.3.2 ¿No pertenece una variable a un intervalo?

Se trata de ver si el valor almacenado una variable está fuera de un intervalo, por ejemplo, $[5, 10]$:

```
# Solución
x <- 7
x < 5 || x > 10
## [1] FALSE
x <- 12
x < 5 || x > 10
## [1] TRUE
```

Otra posibilidad es usar la expresión que comprueba si la variable pertenece al intervalo y negarla.

```
# Solución alternativa
x <- 7
!(x >= 5 && x <= 10)
## [1] FALSE
x <- 12
!(x >= 5 && x <= 10)
## [1] TRUE
```

3.3.3.3 ¿Toma una variable uno de varios posibles valores?

Se trata de ver si el valor almacenado en una variable coincide con alguno de una serie de valores, por ejemplo 2, 7 y 9.

```
x <- 7
x == 2 || x == 7 || x == 9 # ¿vale x 2, 7 o 9?
## [1] TRUE
x <- 12
x == 2 || x == 7 || x == 9 # ¿vale x 2, 7 o 9?
## [1] FALSE
```

Observa que `x == 2 || x == 7 || x == 9` es equivalente a `(x == 2 || x == 7) || x == 9`, porque el operador O es binario y se asocia de izquierda a derecha. Una expresión que tiene varias condiciones O encadenadas es verdadera si, y sólo si, al menos una condición es verdadera (piensa por qué).

Esta condición se puede expresar más fácilmente usando vectores y el operador `%in%`, que comprueba si un elemento está en un vector:

```
x <- 7
x %in% c(2, 7, 9) # ¿está x en el vector [2, 7, 9]?
## [1] TRUE
x <- 12
x %in% c(2, 7, 9) # ¿está x en el vector [2, 7, 9]?
## [1] FALSE
```

3.3.3.4 ¿No toma una variable uno de varios posibles valores?

Se trata de ver si el valor almacenado en una variable no coincide con ninguno de una serie de valores, por ejemplo 2, 7 y 9.

```
x <- 7
x != 2 && x != 7 && x != 9
## [1] FALSE
x <- 12
x != 2 && x != 7 && x != 9
## [1] TRUE
```

Observa que `x != 2 && x != 7 && x != 9` es equivalente a `(x != 2 && x != 7) && x != 9`, porque el operador Y es binario y se asocia de izquierda a derecha. Una expresión que tiene varias condiciones Y encadenadas es verdadera si, y sólo si, todas las condiciones son verdaderas.

Una forma alternativa de expresar la condición es:

```
x <- 7
!(x %in% c(2, 7, 9)) # ¿no está x en el vector [2, 7, 9]?
## [1] FALSE
x <- 12
!(x %in% c(2, 7, 9)) # ¿no está x en el vector [2, 7, 9]?
## [1] TRUE
```

3.4 Invocación a funciones

Como hemos comentado, las funciones sirven para realizar cálculos. Una función es invocada facilitándole unos argumentos. En función de los argumentos realiza

sus cálculos y devuelve un resultado. Por ejemplo, la función `cos` toma como argumento un ángulo en radianes y devuelve su coseno:

```
cos(0)
## [1] 1
cos(pi / 2)
## [1] 6.123032e-17
```

R tiene varias formas de especificar los argumentos de una función al ser invocada. Vamos a estudiarlo con la función `punif` que devuelve un valor de la función de distribución uniforme. Esta función tiene la siguiente cabecera:

- `punif(q, min = 0, max = 1, lower.tail = TRUE, log.p = FALSE)`

Consulta la ayuda de `punif` para una descripción detallada de su funcionamiento. Veamos un ejemplo de invocación:

```
punif(0.75, 0, 1, TRUE, FALSE)
## [1] 0.75
```

Con esta invocación queremos ver la probabilidad de que una variable aleatoria de una distribución uniforme $U(0, 1)$ sea menor o igual que 0.75. Este tipo de invocación es posicional: cada argumento se corresponde con el parámetro que ocupa la misma posición en la cabecera de la función. En el ejemplo, `0.75` se corresponde con `q`, `0` con `min`, `1` con `max` y así sucesivamente. Veamos otro ejemplo con una $U(0, 2)$:

```
punif(0.75, 0, 2, TRUE, FALSE)
## [1] 0.375
```

Si miramos la cabecera de la función veremos que todos los parámetros, salvo el primero, tienen el formato `x = valor`. Esto quiere decir que el parámetro `x` toma el valor por defecto `valor`. El valor por defecto de un parámetro significa que si no se especifica ese argumento en la invocación a la función el parámetro tomará dicho valor por defecto. Por ejemplo:

```
punif(0.75) # equivale a punif(0.75, 0, 1, TRUE, FALSE)
## [1] 0.75
punif(0.75, 0, 2) # equivale a punif(0.75, 0, 2, TRUE, FALSE)
## [1] 0.375
```

Un valor por defecto se elige de forma que sea el valor más común posible. De este modo, en muchas ocasiones no habrá que especificar el argumento.

Observa la segunda llamada: `punif(0.75, 0, 2)`. Aunque 0 es el valor por defecto para el límite inferior de la distribución uniforme, ha habido que especificar dicho argumento porque estamos invocando a la función con argumentos posicionales: `punif(0.75, 2)` equivale a `punif(0.75, 2, 1, TRUE, FALSE)`, que produce un error porque no tiene sentido una distribución $U(2, 1)$.

Afortunadamente, R no obliga a invocar a las funciones con parámetros posicionales. La alternativa es realizar una invocación por nombre, en la que los argumentos se especifican mediante su nombre y el valor que toman:

```
punif(q = 0.75, max = 2) # equivale a punif(0.75, 0, 2, TRUE, FALSE)
## [1] 0.375
punif(max = 2, q = 0.75) # equivale a punif(0.75, 0, 2, TRUE, FALSE)
## [1] 0.375
```

Se puede mezclar invocación posicional con invocación por nombre. En dicho caso, los argumentos posicionales irán primero y se corresponden con sus posiciones en la cabecera. Este tipo de invocación, dada su versatilidad, se usa asiduamente:

```
punif(0.75, max = 2) # uniforme [0, 2]
## [1] 0.375
punif(0.75, lower.tail = FALSE) # equivale a punif(0.75, 0, 1, FALSE, FALSE)
## [1] 0.25
```

La última invocación produce la probabilidad de que una variable aleatoria de una distribución $U(0, 1)$ sea mayor que 0.75.

3.5 Ejercicios

1. Almacena en una variable un precio (por ejemplo, 100) y en otra variable el IVA (por ejemplo, un 21%). Escribe una expresión que calcule el precio al aplicarle el IVA.
2. Escribe las siguientes expresiones aritméticas en R:
   - $\frac{a}{b} + 1$
   - $\frac{a+b}{c+d}$
   - $\frac{a+\frac{b}{c}}{d+\frac{c}{b}}$
   - $a + \frac{b}{c-d}$
   - $(a + b)\frac{c}{d}$

   Prueba que las expresiones son correctas. Para ello asigna a las variables `a`, `b`,`c` y `d` los valores 2, 1, -1 y 4 respectivamente. El resultado de evaluar las distintas expresiones es 3, 1, 0.333, 1.8 y -0.75 respectivamente.

3. Intenta predecir el resultado de evaluar las siguientes expresiones lógicas. Piensa en el orden en que se evalúan las subexpresiones dentro de cada expresión. Los valores de las variables son: `a <- TRUE`, `b <- TRUE`, `c <- FALSE`, `d <- FALSE`.

    - `c || !a && b`
    - `! (a || c) || b && !c`
    - `! (! (! (a && c || d)))`
    - `!(5<3) && a || !(d || c)`

4. Escribe una expresión lógica que compruebe si la variable `x` pertenece al rango $[0, 5]$ o al rango $[10, 15]$.

5. Escribe una expresión lógica que compruebe si una variable contiene un valor par. Nota: el operador `%%` devuelve el resto de la división entera.

6. Utiliza la función `pnorm` para calcular la probabilidad de que una variable de una distribución normal $N(0, 2)$ sea mayor que 3.

7. Calcula algunos datos sobre una circunferencia a partir de su radio.

    - Guarda en la variable radio el valor del radio de la circunferencia con la que vamos a trabajar.
    - Calcula la longitud de la circunferencia de dicho radio.
    - Calcula el área del círculo delimitado por dicha circunferencia.

8. Queremos conocer los datos estadísticos de una asignatura. Para ello necesitamos definir variables que almacenen el número de suspensos, aprobados, notables y sobresalientes de una asignatura (asigna los valores que prefieras). Usa expresiones para calcular:

    - El tanto por ciento de alumnos que han superado la asignatura.
    - El tanto por ciento de suspensos, aprobados, notables y sobresalientes de la asignatura.

Chapter 4

Estructuras secuenciales de un programa

En este tema se estudian las estructuras secuenciales de programación. Estas estructuras se caracterizan porque se ejecutan una detrás de otra, según aparecen escritas en un programa. En el tema previo vimos una de estas estructuras secuenciales: la asignación. En este tema estudiamos el resto: las instrucciones de entrada y de salida. También veremos cómo crear un programa o guion. Un programa es una serie de instrucciones que pueden ser ejecutadas en conjunto.

4.1 Salida de datos

Las instrucciones de salida de datos se utilizan para que el ordenador pueda comunicar información, normalmente los resultados de un cálculo, con el exterior. La información o datos pueden enviarse a distintos dispositivos de salida, como la impresora, el disco duro, la red o la pantalla. Nosotros vamos a enviar información a la pantalla, pero con unos pocos cambios la información podría enviarse a otros dispositivos. Vamos a estudiar dos funciones: `print` y `cat`.

4.1.1 La función `print`

La función `print` toma como parámetro una expresión y muestra en la salida el resultado de evaluar la expresión.

```
x <- 5
print(x ^ 2)
## [1] 25
```

Realmente la salida es la misma que se obtendría si hubiéramos escrito la expresión en la consola. Una ventaja de `print` es que es una función genérica. Esto permite que los programadores escriban versiones específicas de `print` para que trabajen con los objetos que ellos definen. Esto hace que `print` sirva para mostrar en la pantalla casi cualquier objeto de R. A continuación se ajusta un modelo de regresión lineal simple y se usa `print` para mostrar el resultado del ajuste:

```
# Ajusta la longitud del pétalo en función de la anchura del pétalo
modelo <- lm(Petal.Length ~ Petal.Width, data = iris)
print(modelo)
##
## Call:
## lm(formula = Petal.Length ~ Petal.Width, data = iris)
##
## Coefficients:
## (Intercept)  Petal.Width
##       1.084        2.230
```

El `print` de este modelo lineal nos dice que la longitud de pétalo se ajusta como $1.084 + 2.23 Petal.Width$. Aunque `print` es muy flexible, porque permite mostrar objetos de casi cualquier tipo, sólo permite mostrar un objeto por invocación a `print`.

4.1.2 La función `cat`

La función `cat` permite mostrar menos tipos de objetos, pero es posible especificar más de un objeto en una única llamada a la función. Por ejemplo:

```
x <- 5
cat(x, x ^ 2)
## 5 25
```

Los distintos objetos aparecen separados en la salida por un espacio en blanco. En los ejercicios de este tema haremos un uso intensivo de `cat`, mezclando expresiones numéricas con cadenas de caracteres. Una cadena de caracteres es una secuencia de caracteres encerrada entre comillas simples o dobles. En la pantalla, las cadenas de caracteres aparecen con su valor literal.

```
x <- 5
cat(x, "al cuadrado vale", x ^ 2, "\n")
## 5 al cuadrado vale 25
```

En esta llamada a `cat` se han usado dos expresiones numéricas (`x` y `x ^ 2`) y dos cadenas de caracteres: `"al cuadrado vale"` y `"\n"`. En general, `"\n"` será

utilizada como último parámetro de la invocación a `cat`. La cadena `"\n"` produce un salto de línea; si no la utilizamos, las siguientes salidas de invocaciones a `cat` proseguirían en la misma línea que las anteriores. Por ejemplo:

```
cat("Hola"); cat("a todos") # prueba en la consola
## Hola
## a todos
```

Observa que en la consola la segunda salida de `cat` aparece en la misma línea que la primera salida de `cat` y justo detrás. Esto puede ser deseable, pero si queremos que la salida sea en distintas líneas tenemos que usar `"\n"`.

```
cat("Hola", "\n"); cat("a todos")
## Hola
## a todos
```

Realmente `\n` puede utilizarse en cualquier cadena de caracteres para representar un salto de línea, por lo que podemos escribir el código de antes como:

```
cat("Hola\n"); cat("a todos")
## Hola
## a todos
```

O incluso:

```
cat("Hola\na todos")
## Hola
## a todos
```

Si queremos varios saltos de línea usamos `\n` más de una vez dentro de una cadena:

```
cat("Hola\n\n\n"); cat("a todos")
## Hola
## a todos
```

La `\` se conoce como un carácter de escape. Sirve para poder representar en cadenas de caracteres símbolos que no se corresponden con símbolos del teclado. En `\n` la `n` significa *new line*. Otro ejemplo es `\t` que produce una tabulación.

```
cat("a\tb")
## a    b
```

4.2 Entrada de datos

Las instrucciones de entrada de datos permiten que el ordenador se comunique con el exterior para recibir o leer información. La información puede ser leída desde distintos dispositivos como el ratón, el teclado o un archivo. En este apartado vamos a ver una instrucción para leer información del teclado. Realmente, en R no es muy usual leer datos del teclado. Los usuarios de R suelen trabajar con conjuntos de datos que están almacenados en archivos, en internet o que están disponibles directamente en un paquete. Sin embargo, a nosotros nos será útil leer de teclado para aprender a programar de una forma más sencilla y divertida.

La función que vamos a estudiar se llama `readline`. Es una función que, como su nombre indica, lee una línea de texto del teclado. Veamos un ejemplo:

```
# Ejecuta esto en la consola
nombre <- readline("Introduce tu nombre: ")
nombre
```

La función `readline` muestra en la consola el texto que se le pasa como parámetro y lee del teclado una línea de texto terminada por la pulsación de *Enter*. El texto leído del teclado es lo que devuelve la función `readline`. En el ejemplo, el texto leído se asigna a la variable `nombre`.

4.2.1 Lectura de información numérica

La función `readline` devuelve una cadena de caracteres, que es un tipo de datos que almacena una secuencia de caracteres. Posteriormente estudiaremos los distintos tipos de datos de R. Por ahora, nos basta con saber que las cadenas de caracteres no admiten operaciones aritméticas. Para poder trabajar con información numérica hay que usar el tipo de datos `numeric`, que almacena números reales, o `integer` para números enteros. Por lo tanto, si queremos leer y procesar datos numéricos, primero los leeremos como cadenas de caracteres con `readline` y después usaremos una función para convertirlos a reales o enteros. Estas funciones son `as.numeric` y `as.integer` respectivamente. Veamos un ejemplo:

```
# Ejecuta este texto en la consola
x <- readline("Introduce un número: ")
x <- as.numeric(x)
x ^ 2
```

Podemos obtener una versión más corta del código anterior si aplicamos directamente la función `as.numeric` a la salida de `readline`.

```
# Ejecuta este texto en la consola
x <- as.numeric(readline("Introduce un número: "))
x ^ 2
```

4.3 Elaboración de programas o guiones

Ahora que conocemos las instrucciones de entrada y de salida estamos en disposición de escribir programas. Un *programa* o *guion* es una serie de instrucciones que se ejecutan con el objeto de producir un resultado. Normalmente los programas comienzan solicitando datos (entrada), por ejemplo, del teclado. A continuación hacen algunos cálculos para procesar los datos de entrada, generando unos resultados que se muestran en la pantalla u otro dispositivo.

Los programas en R se escriben en un archivo de texto usando un editor de texto. Como nosotros trabajamos en RStudio, usaremos el editor de texto de RStudio. Para crear un nuevo archivo de texto seleccionamos **File > New File > R Script**.

Crea un archivo de texto e introduce el siguiente código, que calcula el área de un rectángulo:

```
b <- as.numeric(readline("Introduce la base: "))
a <- as.numeric(readline("Introduce la altura: "))
cat("El área es", b * a)
```

Guarda el archivo con extensión .R (por ejemplo, `programa.R`). Para ejecutar el programa selecciona la opción **Code > Source**. Observa que la consola se utiliza para introducir y mostrar la información del programa. De cara a ejecutar el programa también puedes usar la opción **Source** en la barra de herramientas del editor de texto.

4.3.1 Instrucciones

Como hemos comentado previamente, un programa consta de una serie de instrucciones, también llamadas sentencias. En general, cada sentencia termina con un carácter de nueva línea o salto de línea, pero si una sentencia es larga puede ocupar varias líneas. Si queremos escribir varias sentencias en una misma línea hay que separar las sentencias con el carácter punto y coma. Por ejemplo:

```
x <- 2; y <- 6.5 # línea con dos sentencias de asignación
print(x * y)
## [1] 13
```

4.4 Comentarios

En fragmentos de código previos ya se han usado comentarios. Un comentario es un texto que sirve para aclarar cosas del código. Los comentarios no son ejecutados, simplemente sirven para facilitar la comprensión de un programa a alguien que lee su código. En R un comentario comienza por el carácter `#` y va hasta el final de la línea en que se encuentra dicho carácter. Vamos a comentar el programa previo:

```
# Programa: Área de un rectángulo
# Entradas: la base y altura del rectángulo
# Salidas: el área del rectángulo
b <- as.numeric(readline("Introduce la base: "))
a <- as.numeric(readline("Introduce la altura: "))
cat("El área es", b * a)
```

4.5 Ejercicios

1. Escribe un programa que lea dos números y muestre en la salida su suma, resta, multiplicación y división. Suponiendo que el usuario introduce los números 6 y 3 el programa mostrará lo siguiente:

   Suma: 9
   Resta: 3
   Producto: 18
   División: 2

2. Escribe un programa que solicite los catetos de un triángulo rectángulo y muestre su hipotenusa. Caso de prueba: Longitud de los catetos (3 y 4), hipotenusa (5).

3. La calificación final de un estudiante es la media ponderada de tres notas: la nota de prácticas que cuenta un 30% del total, la nota teórica que cuenta un 60% y la nota de participación que cuenta el 10% restante. Escribe un programa que lea las tres notas de un alumno y escriba en la pantalla su nota final. Puedes probar este programa con los siguientes datos: nota de prácticas (5), nota de teoría (7) y nota de participación (10). La calificación final para estos datos es 6.7.

4. Escribe un programa que lea la nota final de cuatro alumnos y calcule la nota final media de los cuatro alumnos. Para probarlo, dadas las notas: 5.6, 6, 10 y 9, su nota media es 7.65.

5. Realiza un programa que calcule el valor que toma la siguiente función para unos valores dados de `x` e `y`:

$$f(x,y) = \frac{\sqrt{x}}{y^2 - 1}$$

Caso de prueba: $x = 10$, $y = 3$, salida $f(x,y) = 0.395$.

6. Escribe un programa que calcule las soluciones de una ecuación de segundo grado de la forma $ax^2 + bx + c = 0$, teniendo en cuenta que:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Supón que a es distinto de cero. Como caso de prueba, dada $2x^2+9x+3 = 0$, sus raíces son -0.36 y -4.14.

7. Suponiendo que el recibo de la luz sube un 3% cada año, realice un programa que solicite una factura de este año y una cantidad de años y muestre en la salida cuánto valdría la factura dentro del número de años introducidos. Caso de prueba: factura de 100 euros y 3 años, la factura dentro de 3 años será 109.273.

8. Escribe un programa que calcule la desviación estándar de una muestra con 5 elementos:

$$\sigma = \sqrt{\frac{1}{4}\sum_{i=1}^{5}(x_i - \overline{x})^2}$$

Caso de prueba: 1, 4.5, 7, 3 y 15, desviación estándar: 5.436.

9. Pide al usuario una cantidad en segundos y pásala a horas, minutos y segundos. Para resolver este ejercicio ten en cuenta que el número de segundos es un valor entero. En R, el operador `%/%` es la división entera y el operador `%%` es el resto de la división entera. Por ejemplo: `13 %/% 4` vale 3 y `13 %% 4` vale 1.

10. Escribe un programa que asigne valores a dos variables y después intercambie los valores de las dos variables. Sugerencia: apóyese en una variable auxiliar.

Chapter 5

Tipos de datos: cadenas de caracteres, vectores y factores

En este tema y el siguiente se estudian los principales tipos de datos de R. Todo dato tiene que tener asociado un tipo. Según el tipo del dato se utilizará una determinada representación para almacenar el dato en el ordenador. El tipo de un dato también determina qué operaciones es posible realizar con él. Por ejemplo, es posible sumar números o matrices numéricas, pero no secuencias de caracteres. En este tema vamos a estudiar los tipos de datos: cadena de caracteres, vector y factor.

5.1 Cadenas de caracteres

En temas previos ya hemos trabajado con tipos de datos básicos como números y valores lógicos. Las cadenas de caracteres también son un tipo de dato básico. Consisten en una secuencia de caracteres y sirven para representar información de tipo texto, como el nombre de una persona, el nombre de una calle, una matrícula de un coche, etcétera.

En R una cadena de caracteres se representa como una serie de caracteres encerradas entre comillas dobles o simples:

```
nombre <- "Diego"        # usando comillas dobles
matricula <- '1234 ABC'  # usando comillas simples
```

Independientemente de que usemos comillas dobles o simples, R nos mostrará

la cadena en la consola con comillas dobles o sin ningún tipo de entrecomillado, pero nunca con comillas simples.

Si es necesario usar un tipo de comillas en una cadena de caracteres, podemos usar el otro tipo de comillas para delimitar la cadena:

```
'Me dijo: "detente"'
## [1] "Me dijo: \"detente\""
"Me dijo: 'detente'"
## [1] "Me dijo: 'detente'"
```

Otra posibilidad es usar una secuencia de escape con el carácter de escape \:

```
"Me dijo: \"detente\""
## [1] "Me dijo: \"detente\""
```

Otras secuencias de escape muy utilizas son: \\ (barra invertida), \n (salto de línea) y \t (tabulador).

Existen funciones como `paste`, `paste0` o `substr` que permiten trabajar con cadenas de caracteres, aunque no las vamos a estudiar aquí. Estas funciones son muy versátiles y, dependiendo de cómo se invoquen, son capaces de realizar distintos procesamientos. Por ejemplo, con `paste` se puede obtener una cadena de caracteres mezclando información de tipo texto con numérica, trabajando de una forma parecida a `cat`.

```
ang <- 0
cadena <- paste("El coseno de", ang, "vale", cos(ang))
cadena
## [1] "El coseno de 0 vale 1"
```

5.2 Vectores

Un vector es una colección ordenada de datos del mismo tipo. Es posible acceder a los elementos de un vector individual o parcialmente. Los vectores son uno de los tipos de datos más importantes de R y de la mayoría de los lenguajes de programación.

5.2.1 Creación de vectores

En R es posible crear vectores de un gran número de formas. Una de las más utilizadas es usando la función `c`, que toma como parámetros un número arbitrario de elementos. La función `c` devuelve un vector con la concatenación de dichos elementos.

```
v <- c(2, 8.3, 5.4) # vector numérico formado por 3 números
v
## [1] 2.0 8.3 5.4
```

La función `c` admite vectores como parámetros:

```
v1 <- c(1, 3)
v2 <- c(7, 9)
v <- c(v1, 5, v2)
v
## [1] 1 3 5 7 9
```

Los elementos de un vector pueden ser de cualquier tipo base:

```
dedos <- c('Pulgar', 'Índice', 'Corazón', 'Anular', 'Meñique')
dedos
## [1] "Pulgar"  "Índice"  "Corazón" "Anular"  "Meñique"
(v <- c(TRUE, FALSE, FALSE)) # vector de valores lógicos
## [1]  TRUE FALSE FALSE
```

5.2.1.1 Generación de secuencias regulares

El operador `:` permite generar un vector que almacena una secuencia de enteros consecutivos. Por ejemplo `1:5` equivale a `c(1, 2, 3, 4, 5)`. También es posible generar secuencias en orden inverso:

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

El operador `:` tiene una prioridad alta, mayor que la de algunos operadores aritméticos:

```
n <- 10
5:n-1   # equivale a (5:n)-1
## [1] 4 5 6 7 8 9
5:(n-1) # 5:9
## [1] 5 6 7 8 9
```

La función `seq` también permite generar secuencias numéricas. `seq(1, 10)` equivale a `1:10`. Sin embargo, `seq` tiene parámetros con nombre que permiten generar otro tipo de secuencias. Por ejemplo, `by` especifica la distancia entre los elementos:

```
seq(10, 12, by = .2)
##  [1] 10.0 10.2 10.4 10.6 10.8 11.0 11.2 11.4 11.6 11.8 12.0
seq(from = 11, to = 9, by = -.5)
## [1] 11.0 10.5 10.0  9.5  9.0
```

El parámetro con nombre `length.out` también es muy usado. Indica la longitud de la secuencia:

```
seq(0, pi, length.out = 10)
##  [1] 0.0000000 0.3490659 0.6981317 1.0471976 1.3962634 1.7453293 2.0943951
##  [8] 2.4434610 2.7925268 3.1415927
```

Esta última invocación genera 10 números equidistantes en el intervalo $[0, \pi]$ (los extremos del intervalo se incluyen en la secuencia).

Otra función muy útil es `rep`, que sirve para replicar valores de un vector. Veamos algunos ejemplos:

```
rep(c(2, 1, 3), times = 4)
##  [1] 2 1 3 2 1 3 2 1 3 2 1 3
rep(c(2, 1, 3), each = 4)
##  [1] 2 2 2 2 1 1 1 1 3 3 3 3
rep(c("a", "b", "c"), times = c(2, 1, 5))
## [1] "a" "a" "b" "c" "c" "c" "c" "c"
```

5.2.1.2 Vector vacío

Aunque no sea algo frecuente, a veces se necesita generar un vector vacío, es decir, sin datos. A continuación se ilustra una forma de hacerlo:

```
v <- numeric()
length(v)
## [1] 0
```

La función `length`, aplicada a un vector, devuelve cuántos elementos tiene. El ejemplo anterior sirve para un vector de datos reales, para datos de distintos tipos básicos habría que cambiar `numeric` por:

- `integer` (enteros)
- `character` (cadenas de caracteres)
- `logical` (valores lógicos)
- `complex` (números complejos)

Otra posibilidad es usar la función `vector` (consulta su ayuda si estás interesado en ver las distintas opciones):

```
v <- vector()
length(v)
## [1] 0
```

5.2.1.3 Generación de números aleatorios

Existen muchas funciones que devuelven sus resultados como un vector. Entre ellas se encuentran las funciones generadoras de números aleatorios, como `runif` o `rnorm`. Por ejemplo:

```
runif(10)
##  [1] 0.31999450 0.36210054 0.26640064 0.68464443 0.93009203 0.38021304
##  [7] 0.19167845 0.21461285 0.92389324 0.04791454
rnorm(5)
## [1]  0.47587831  0.01931357 -0.10410679  0.90435457 -0.11322213
```

La primera genera una muestra aleatoria de variables independientes de una $U(0,1)$ y la segunda de una $N(0,1)$. Por supuesto, se puede especificar los parámetros de la distribución (usa la ayuda de estas funciones para ver cómo se hace).

La generación de números aleatorios juega un papel crucial en el desarrollo de simulaciones con ordenador. Sin embargo, la aleatoriedad de los números generados hace que distintas ejecuciones de una simulación produzcan distintos resultados. Aunque esto es lo deseable, algunas veces puede dificultar la realización de un programa que implementa una simulación. Puesto que los números aleatorios generados son realmente pseudoaleatorios (generan una secuencia determinista a partir de una semilla), en la fase de desarrollo de una simulación podemos establecer la semilla para obtener siempre los mismos resultados. La función `set.seed` permite especificar la semilla. Veamos un ejemplo:

```
runif(5) # 5 números U(0, 1)
## [1] 0.4268347 0.8900439 0.2310605 0.6823289 0.4774462
runif(5) # 5 números U(0, 1)
## [1] 0.7958001 0.9497106 0.3968653 0.2105817 0.8125346
set.seed(8)
runif(5) # muestra de 5 U(0, 1) a partir de semilla 8
## [1] 0.4662952 0.2078233 0.7996580 0.6518713 0.3215092
set.seed(8)
runif(5) # La misma muestra que antes
## [1] 0.4662952 0.2078233 0.7996580 0.6518713 0.3215092
```

Otra función interesante es `sample`, que obtiene una muestra aleatoria de un vector de elementos. Por defecto, la muestra es sin reemplazo y todos los elementos tienen la misma probabilidad de salir, pero hay parámetros para modificar este comportamiento por defecto.

```
sample(1:10, 5) # muestra (sin reemplazo) de tamaño 5 de números del 1 al 10
## [1]  7 10  6  1  5
sample(c("a", "b", "c"), 5, replace = TRUE) # con reemplazo
## [1] "b" "c" "c" "b" "b"
```

`sample(n)` genera una permutación aleatoria de los valores en `1:n`.

5.2.1.4 Lectura de datos desde el teclado

La función `scan` permite leer una serie de valores almacenados en un archivo o, en su defecto, provenientes del teclado. `scan` tiene muchas opciones para controlar la lectura. Usada sin parámetros lee del teclado números separados por espacios en blancos y/o saltos de línea. La lectura termina cuando se introduce una línea en blanco y los número leídos se devuelven en un vector:

```
v <- scan() # prueba a ejecutarlo en la consola
```

5.2.2 Tipo de un vector

Se puede obtener el tipo de un vector (de sus elementos) con la función `typeof`:

```
v1 <- c(1, 2.8, -3.5)
typeof(v1)
## [1] "double"
v2 <- c(2L, 20L)
typeof(v2)
## [1] "integer"
v3 <- c(TRUE, FALSE, FALSE)
typeof(v3)
## [1] "logical"
v4 <- c("1234 ABC", "2222 JKF")
typeof(v4)
## [1] "character"
```

Un vector puede almacenar elementos de tipo *double* (real), *integer* (entero), *logical* (lógico), *character* (cadena de caracteres), *complex* (complejo) y *raw*. Los dos últimos tipos no se usan demasiado, especialmente *raw*, y no los utilizaremos en estos apuntes.

Se puede comprobar si un vector es de un tipo concreto con las funciones `is.double`, `is.integer`, `is.logical`, `is.character`, `is.complex` e `is.raw`:

```
is.logical(v4)
## [1] FALSE
is.character(v4)
## [1] TRUE
```

Todos los elementos de un vector deben ser del mismo tipo. Si se intenta crear un vector con elementos de distinto tipo, todos los elementos se *convierten* al tipo con mayor rango de representación. El rango de menor a mayor es: lógico, entero, real, carácter. Veamos algún ejemplo:

```
c(TRUE, 9)      # se convierten todos a entero
## [1] 1 9
c(5.4, "diez")  # se convierten todos a cadena
## [1] "5.4"  "diez"
```

Cuando los valores lógicos se convierten a números `FALSE` se convierte en 0 y `TRUE` en 1. Se puede forzar la conversión de un vector de un tipo a otro con las funciones `as.TIPO`:

```
v <- c(TRUE, FALSE, TRUE)
as.integer(v)
## [1] 1 0 1
as.integer(c("1", "1.5", "a"))
## Warning: NAs introducidos por coerción
## [1]  1  1 NA
```

En la última conversión `as.integer` no sabe cómo convertir `"a"` a entero, por lo que genera el valor `NA` y muestra una advertencia.

Como ejercicio, intenta predecir qué producirán las siguientes expresiones: `c(1, FALSE)`, `c("a", 7)` y `c(TRUE, 1L)`. Escríbelas y comprueba si has acertado.

5.2.3 Aritmética de vectores

Una de las ventajas de R frente a otros lenguajes de programación es que en R se puede aplicar los operadores aritméticos (`+`, `-`, `*`, etcétera) a los vectores (los vectores deberán ser de algún tipo numérico o de tipo lógico):

```
v1 <- c(-5, 2, 8.1)
v2 <- c(4, -3, 9)
v1 + v2
## [1] -1.0 -1.0 17.1
```

En caso de que los vectores no tengan la misma longitud, el resultado será de la longitud del vector más largo y los vectores más pequeños se *reciclan* para que encajen con la longitud del vector más largo.

```
v1 <- c(-5, 2, 8.1)
v2 <- 1:2
v1 + v2  # v2 se recicla a c(1, 2, 1)
## Warning in v1 + v2: longitud de objeto mayor no es múltiplo de la longitud de
## uno menor
## [1] -4.0  4.0  9.1
```

Aunque el código previo es correcto, R nos avisa porque la longitud de `v1` no es un múltiplo de la de `v2`. En el siguiente ejemplo R no avisa:

```
v1 <- c(-5, 2, 8.1, 2)
v2 <- 1:2
v1 + v2  # v2 se recicla a c(1, 2, 1, 2)
## [1] -4.0  4.0  9.1  4.0
```

Es interesante saber que en R cuando escribimos un valor literal, como 7, realmente se trata como un vector de un elemento (es decir, como `c(7)`). Por lo tanto, lo siguiente es válido:

```
x <- 1:10
2 * x  - 3
##  [1] -1  1  3  5  7  9 11 13 15 17
```

Realmente 2 es un vector de longitud uno que se recicla para que tenga la longitud de `x` (es decir, se recicla a un vector con 10 doses), y lo mismo ocurre con 3. El efecto del código previo es equivalente a:

```
x <- 1:10
rep(2, times = length(x)) * x - rep(3, times = length(x))
##  [1] -1  1  3  5  7  9 11 13 15 17
```

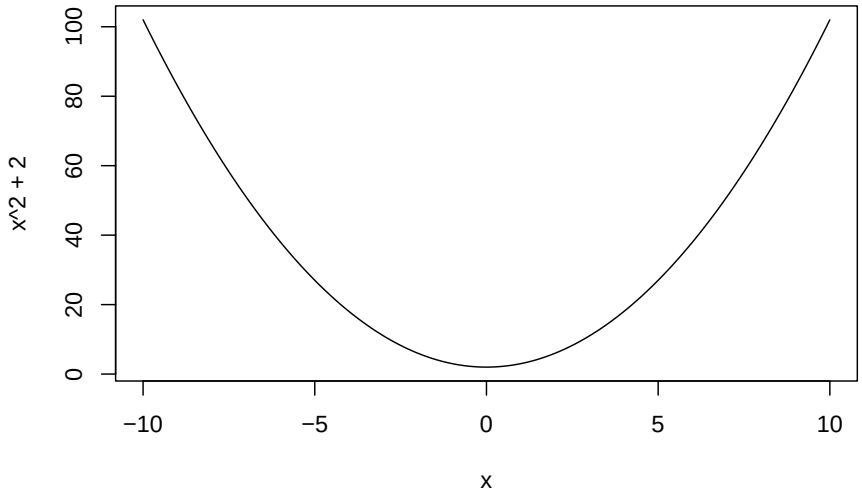
Las funciones como `sqrt`, `log`, `exp`, `cos`, `sin`, `abs`, `round`, ... están escritas para trabajar con vectores:

```
exp(seq(1, 7, by = 2) / 2)
## [1]  1.648721  4.481689 12.182494 33.115452
```

produce el vector c($e^{1/2}$, $e^{3/2}$, $e^{5/2}$, $e^{7/2}$).

La combinación de aritmética de vectores con funciones gráficas, como `plot`, que estudiaremos más adelante, permite visualizar de una forma sencilla funciones matemáticas:

```
# Función x^2 + 2 en el intervalo [-10, 10]
x <- seq(-10, 10, by = .1)
plot(x, x^2 + 2, type = "l")
```



5.2.4 Funciones aplicables a vectores

Aparte de las funciones vistas anteriormente, como `exp`, que aplican una operación a los distintos elementos de un vector, hay otras funciones que realizan un procesamiento sobre los elementos de un vector en conjunto. La lista es muy grande, por lo que vamos a citar sólo algunas:

- `mean`, `median`, `var` y `sd`: calculan la media, mediana, varianza muestral y desviación típica muestral respectivamente de una muestra de valores.
- `max` y `min`: el máximo y mínimo repectivamente.
- `range`: produce el vector `c(min(v), max(v))`, donde `v` es el vector al que se aplica.
- `sum` y `prod`: calculan la sumatoria y el productorio de los elementos de un vector respectivamente.
- `cumsum` y `cumprod`: suma y producto acumulado respectivamente.
- `sort`: ordena los elementos (por defecto, en orden creciente).
- `rev`: devuelve el vector en orden inverso.

Veamos algunos ejemplos:

```
x <- c(2, 9, 5, 2.3)
median(x)
## [1] 3.65
max(x)
## [1] 9
sum(x)
## [1] 18.3
cumsum(x)
## [1]  2.0 11.0 16.0 18.3
sort(x)
## [1] 2.0 2.3 5.0 9.0
rev(x)
## [1] 2.3 5.0 9.0 2.0
cumprod(1:5) # factoriales de los números del 1 al 5
## [1]   1   2   6  24 120
```

La aritmética de vectores, junto con el hecho de que gran parte de las funciones se puedan aplicar a vectores permite expresar cálculos complejos de una forma sencilla. Por ejemplo, la función `sd` calcula la desviación típica de una muestra `x` formada por n números según la fórmula:

$$\sigma = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(x_i - \overline{x})^2}$$

En caso de que no existiera la función `sd`, se podría escribir de una forma relativamente sencilla usando aritmética de vectores:

```
v <- runif(1000, max = 6) # muestra de tamaño 1000 de una U(0, 6)
sd(v)                     # desviación estándar usando sd
## [1] 1.748962
suma <- sum((v - mean(v)) ^ 2) # sumatorio
sigma <- sqrt(suma/(length(v) - 1))
sigma
## [1] 1.748962
```

5.2.5 Vectores lógicos

Un vector lógico es aquel cuyos elementos son valores lógicos, por ejemplo:

```
v <- c(TRUE, TRUE, FALSE)
v
## [1]  TRUE  TRUE FALSE
```

Los operadores relacionales (`<`, `>`, `<=`, `>=`, `==`, `!=`) se aplican de una forma natural a los vectores, produciendo como resultado un vector lógico:

```
(v <- rnorm(5))
## [1]  1.52707280  0.25230838 -0.08592673  1.44405132  0.02543926
v > 0
## [1]  TRUE  TRUE FALSE  TRUE  TRUE
"casa" == c("barco", "casa", "pez")
## [1] FALSE  TRUE FALSE
```

Observa que en las expresiones previas `0` y `casa` se reciclan para coincidir con las longitudes de los distintos vectores. En el siguiente caso, los dos vectores tienen la misma longitud:

```
(v1 <- runif(4))
## [1] 0.9147716 0.9158568 0.8826998 0.7334208
(v2 <- runif(4))
## [1] 0.09659358 0.24342786 0.23685341 0.63583959
v1 >= v2
## [1] TRUE TRUE TRUE TRUE
```

Cuando un vector lógico es aplicado en una operación aritmética, los valores `FALSE` se convierten a 0 y los `TRUE` a 1. Esto permite aplicar `sum` y `mean` a vectores lógicos para calcular cuántos elementos de un vector (o qué proporción) verifican una condición:

```
x <- c(-2, 3, 4, -1, 4.2)
(condicion <- x > 0)
## [1] FALSE  TRUE  TRUE FALSE  TRUE
sum(condicion) # ¿Cuántos elementos son mayores que cero?
## [1] 3
sum(x > 0)  # En una única expresión
## [1] 3
mean(x > 0) # Proporción de elementos mayores que cero
## [1] 0.6
```

5.2.5.1 Operadores lógicos

En un tema previo vimos los operadores lógicos: Y (`&&`), O (`||`) y NO (`!`). Los operadores lógicos se pueden aplicar a vectores lógicos. Cuando se aplica a vectores lógicos, el operador lógico Y cambia de `&&` a `&`. El operador O cambia de `||` a `|`. Estos operadores se aplican elemento a elemento. Por ejemplo:

```
v1 <- c(TRUE, FALSE, TRUE, FALSE)
v2 <- c(TRUE, TRUE, FALSE, FALSE)
v1 & v2
## [1]  TRUE FALSE FALSE FALSE
v1 | v2
## [1]  TRUE  TRUE  TRUE FALSE
!v1
## [1] FALSE  TRUE FALSE  TRUE
```

Como veremos más adelante, las expresiones lógicas combinadas con la indexación lógica, resultan muy útiles para trabajar con vectores. Veamos algún otro ejemplo de expresiones lógicas:

```
v <- c(2, 5, 9, 4, 6)
v >= 5 & v <= 10 # ¿v en el rango [5, 10]?
## [1] FALSE  TRUE  TRUE FALSE  TRUE
sum(v >= 5 & v <= 10) # Cantidad de valores en [5, 10]
## [1] 3
mean(v >= 5 & v <= 10) # Proporción de valores en [5, 10]
## [1] 0.6
```

Existen dos funciones muy útiles para trabajar con vectores lógicos. Se trata de `all`, que indica si todos los elementos de un vector lógico son verdaderos y `any` que indica si al menos uno es verdadero.

```
v <- c(2, 4, 8, 10, 11)
v %% 2 == 0      # Calcula si los elementos de v son pares o no
## [1]  TRUE  TRUE  TRUE  TRUE FALSE
all(v %% 2 == 0) # ¿Todos los elementos de v son pares?
## [1] FALSE
any(v %% 2 == 0) # ¿Algún elemento de v es par?
## [1] TRUE
all(v > 0)       # ¿Son todos positivos?
## [1] TRUE
```

Otra función relacionada con vectores lógicos es `which`. Esta función devuelve los índices de un vector lógico con valores a verdadero:

```
(v <- sample(10))
##  [1]  2 10  9  3  6  8  1  7  4  5
which(v > 5) # índices de v con valores mayores que 5
## [1] 2 3 5 6 8
which(v == max(v)) # índice del máximo en v
## [1] 2
```

```
n <- 20
which(n %% 1:n == 0) # divisores del número n
## [1]  1  2  4  5 10 20
```

Por último, la función `xor` calcula la función O exclusiva. Esta función lógica indica si sólo uno de sus dos operandos es verdadero.

```
v1 <- c(TRUE, FALSE, TRUE, FALSE)
v2 <- c(TRUE, TRUE, FALSE, FALSE)
xor(v1, v2)
## [1] FALSE  TRUE  TRUE FALSE
```

No existe un operador O exclusivo, porque la operación `xor` se puede obtener mediante los operadores básicos: O, Y, NO. Piensa un poco e intenta expresar `xor` mediante estos operadores. La solución es la siguiente:

```
v1 <- c(TRUE, FALSE, TRUE, FALSE)
v2 <- c(TRUE, TRUE, FALSE, FALSE)
(v1 & !v2) | (!v1 & v2)
## [1] FALSE  TRUE  TRUE FALSE
```

5.2.6 Indexación de un vector

A veces es útil trabajar con una parte de los elementos de un vector, ya sea para consultar sus valores o para modificarlos. La indexación permite seleccionar una parte de los elementos de un vector. Los vectores de R admiten cuatro tipos de indexación, que describimos en las siguientes subsecciones. El operador de indexación es `[]`. Para indexar hay que especificar un vector de índices encerrado entre `[]`.

5.2.6.1 Mediante un vector de enteros positivos

En este caso los valores del vector de índices deben pertenecer al conjunto $\{1, 2, ..., length(v)\}$, donde v es el nombre del vector que se indexa. Por ejemplo:

```
v <- c(1, 4, 9, 16, 25)
v[c(2, 5)] # índices 2 y 5
## [1]  4 25
v[1:3]     # 3 primeros elementos (índices del 1 al 3)
## [1] 1 4 9
v[c(1, 2, 2, 4)] # se pueden repetir valores
## [1]  1  4  4 16
```

Observa que el primer índice de un vector en R es 1, en otros lenguajes de programación es 0. Cuando sólo se quiere acceder a un elemento de un vector se puede usar la siguiente sintaxis:

```
v <- c(1, 4, 9, 16, 25)
v[3]  # tercer elemento, equivale a v[c(3)]
## [1] 9
```

`v[3]` equivale a `v[c(3)]` porque realmente `3` equivale a `c(3)`. En los ejemplos anteriores hemos utilizado la indexación para consultar los elementos de un vector, pero también puede usarse para modificar sus elementos:

```
v <- c(1, 4, 9, 16, 25)
v[3] <- 0
v
## [1]  1  4  0 16 25
v[1:2] <- 0  # aquí 0 se recicla a c(0, 0)
v
## [1]  0  0  0 16 25
v[1:3] <- c(4, 10, -1)
v
## [1]  4 10 -1 16 25
```

5.2.6.2 Mediante un vector de enteros negativos

Es similar a lo visto en el apartado anterior, pero los valores del vector de índices son negativos e indican los índices a excluir, en lugar de los índices a incluir:

```
v <- c("a", "e", "i", "o", "u")
v[-c(2,3)] # todos los índices, salvo el 2 y el 3
## [1] "a" "o" "u"
v[c(-2, -3)] # igual al anterior
## [1] "a" "o" "u"
v[-length(v)] # todos salvo el último
## [1] "a" "e" "i" "o"
```

5.2.6.3 Mediante un vector de valores lógicos

En este caso el vector contiene valores lógicos. Los índices correspondientes a valores verdaderos en el vector lógico son seleccionados, los que son falsos no.

```
v <- c("Juan", "Pascal", "Julio")
soltero <- c(FALSE, TRUE, TRUE)
```

```
v[soltero]
## [1] "Pascal" "Julio"
```

En el caso de que la longitud del vector lógico sea inferior a la del vector indexado, el vector lógico es reciclado:

```
v <- (1:7) ^ 2
v
## [1]  1  4  9 16 25 36 49
v[c(TRUE, FALSE)] # posiciones impares del vector
## [1]  1  9 25 49
```

El último ejemplo usa el reciclado para seleccionar los elementos del vector que ocupan índices impares.

Se puede usar el índexado lógico para seleccionar los elementos de un vector que verifican una condición. Esta operación es muy útil y habitual y a veces es conocida como *filtrado*:

```
v <- rnorm(10)
v
##  [1] -3.28193174  0.07959148  0.28338913 -1.13800059  0.55833481  1.45478371
##  [7]  0.91997026 -2.45861485  1.35749161 -0.73312796
positivos <- v[v > 0]
positivos
## [1] 0.07959148 0.28338913 0.55833481 1.45478371 0.91997026 1.35749161
v2 <- v[v >= -1.5 & v <= 1.5] # valores en el rango [-1.5, 1.5]
v2
## [1]  0.07959148  0.28338913 -1.13800059  0.55833481  1.45478371  0.91997026
## [7]  1.35749161 -0.73312796
v[v < 0] <- -v[v < 0] # convierte en positivos los negativos
v
##  [1] 3.28193174 0.07959148 0.28338913 1.13800059 0.55833481 1.45478371
##  [7] 0.91997026 2.45861485 1.35749161 0.73312796
```

El último procesamiento también se puede hacer con la instrucción `v <- abs(v)`.

5.2.6.4 Mediante un vector de cadenas de caracteres

Para usar esta posibilidad el vector debe poseer un atributo *names* que permita identificar a sus componentes. Vamos a ver cómo se especifica este atributo:

```
edades <- c(18, 17, 18, 19)
names(edades) <- c("Ana", "Sara", "Juan", "Julia")
edades
##   Ana  Sara  Juan Julia
##    18    17    18    19
```

El indexado con cadenas es similar al indexado con valores enteros positivos, pero usando un vector de cadenas de caracteres:

```
edades[c("Ana", "Julia")]
##   Ana Julia
##    18    19
edades["Ana"] <- edades["Ana"] + 1 # incrementa la edad
```

Es posible crear un vector con nombres usando la función `c`:

```
edades <- c(Ana = 18, Sara = 17, "Juan Pedro" = 18, Luis = 19)
edades
##        Ana       Sara Juan Pedro       Luis
##         18         17         18         19
```

5.2.7 Valores perdidos

Un valor perdido (*missing value*) es un valor no disponible. Esto ocurre con frecuencia en estadística porque se desconoce el valor de una variable, por ejemplo, la edad de una persona en una encuesta (esto podría ocurrir porque el encuestado no la introdujo o porque no se entiende lo que escribió). En R se utiliza la palabra reservada `NA` (*Not available*) para indicar un valor perdido. Hay que tener en cuenta que la mayoría de operaciones que implican un valor perdido producen un valor perdido. Por ejemplo:

```
pesos <- c(77, 68.2, 90.5, NA, 61.5)
pesos_en_libras <- pesos * 2.205
pesos_en_libras
## [1] 169.7850 150.3810 199.5525       NA 135.6075
mean(pesos)
## [1] NA
```

Esto es lógico, porque, por ejemplo, no podemos saber la media de un conjunto de valores si se desconoce el valor de algún elemento.

Si queremos eliminar los valores perdidos podemos usar la función `is.na`, que aplicada a un vector produce un vector lógico indicando si los elementos del vector son valores perdidos:

```
is.na(pesos)
## [1] FALSE FALSE FALSE  TRUE FALSE
```

Ahora la media de los pesos conocidos puede calcularse como:

```
mean(pesos[!is.na(pesos)])
## [1] 74.3
```

Algunas funciones como `mean` o `sd` tienen un parámetro con nombre para eliminar los valores perdidos del cómputo:

```
mean(pesos, na.rm = TRUE)
## [1] 74.3
```

Es tentador usar `pesos == NA` en lugar de `is.na(pesos)`, pero la expresión `pesos == NA` produce `c(NA, NA, NA, NA, NA)`. Esto es correcto, aunque sorprenda, porque no puedes saber si un valor no disponible coincide con otro valor (disponible o no).

Hay que tener en cuenta que hay un segundo tipo de valor que es considerado como un valor perdido por la función `is.na`, se trata del valor `NaN` (*Not a Number*). Este valor se genera al realizar ciertas operaciones aritméticas indefinidas:

```
0 / 0
## [1] NaN
Inf - Inf # Inf significa infinito
## [1] NaN
```

Para distinguir entre `NA` y `NaN` ten en cuenta que `is.na` es cierto para ambos valores, pero `is.nan` es válido sólo para `NaN`.

> **Ejercicio**: Dado el vector especificado más abajo, calcula los índices de dicho vector con valores `NA`.

```
set.seed(10)
(v <- sample(c(NA, NaN, 1), size = 6, replace = TRUE))
## [1]   1  NA NaN   1 NaN   1
```

```
# Solución:
which(is.na(v) & !is.nan(v))
## [1] 2
```

5.2.8 Crecimiento dinámico de un vector

La mayoría de las veces un vector se creará con un tamaño y éste no se modificará. Sin embargo, un vector puede crecer o decrecer. Vamos a verlo con un ejemplo:

```
(v <- c(3, 2))
## [1] 3 2
v[5] <- 11     # v crece de tamaño 2 a 5
v              # los valores indefinidos valen NA
## [1]  3  2 NA NA 11
length(v) <- 3 # ahora el vector decrece
v
## [1]  3  2 NA
```

En concreto, para añadir un elemento al final de un vector tenemos varias posibilidades:

```
v <- 1:3
v <- c(v, 20)          # forma 1: añade 20 al final del vector
v[length(v) + 1] <- 30 # forma 2: añade 30 al final del vector
v
## [1]  1  2  3 20 30
```

5.2.9 Operaciones con conjuntos

Un vector sin repetidos permite representar el concepto matemático de un conjunto (una colección de elementos sin repetidos y sin un orden determinado). R tiene implementadas las operaciones básicas entre conjuntos, como unión, intersección, etcétera.

```
x <- c(1, 2, 5)
y <- c(5, 6, 1, 3)
union(x, y)      # unión
## [1] 1 2 5 6 3
intersect(x, y)  # intersección
## [1] 1 5
setdiff(x, y)    # diferencia: elementos de x que no están en y
## [1] 2
setdiff(y, x)
## [1] 6 3
setequal(x, y)   # igualdad
## [1] FALSE
setequal(x, c(5, 2, 1))
## [1] TRUE
```

Puedes observar que `setequal` no tiene en cuenta el orden de los elementos, puesto que, por definición de conjunto, los elementos de un conjunto no tienen un orden definido.

Para comprobar si un conjunto contiene un elemento se puede utilizar el operador `%in%` o la función `is.element`:

```
2 %in% x
## [1] TRUE
c(2, 4) %in% x # comprobamos dos elementos
## [1]  TRUE FALSE
is.element(2, x)
## [1] TRUE
is.element(c(2, 4), x)
## [1]  TRUE FALSE
```

5.3 Factores

Un factor es un vector utilizado para especificar una clasificación discreta (agrupamiento) de los componentes de otros vectores de la misma longitud. Es decir, los factores sirven para representar datos categóricos. En principio, esta información se puede guardar con cadenas de caracteres, con enteros o valores lógicos (dependiento del tipo de la información), pero usar factores tiene alguna ventaja. Como ejemplo, vamos a suponer que tenemos un vector con las notas de un examen de prácticas. El grupo de prácticas de los alumnos se guarda en otro vector:

```
notas <- c(10, 7, 6, 8)
grupo <- c("g1", "g2", "g2", "g2") # dato categórico
```

Es decir, el primer alumno sacó un 10 y pertenece al grupo 1, el segundo sacó un 7 y pertenece al grupo 2, etcétera. En este ejemplo el grupo de un alumno es un dato categórico. Un pequeño problema de la representación del dato categórico con un vector de cadenas de caracteres es que no sabemos cuántas categorías hay. Viendo el vector se observan dos grupos, pero ¿y si hay un grupo en el que nadie se ha presentado al examen? Los factores permiten especificar todos los valores posibles de la categoría:

```
notas <- c(10, 7, 6, 8)
grupo <- c("g1", "g2", "g2", "g2")
grupo2 <- factor(c("g1", "g2", "g2", "g2"), levels = c("g1", "g2", "g3"))
grupo2
## [1] g1 g2 g2 g2
## Levels: g1 g2 g3
```

Además, al visualizar el factor se puede ver todos los posibles valores o niveles de la categoría.

La función `table` se aplica a un factor y cuenta ocurrencias de cada categoría (tabla de frecuencias):

```
table(grupo)
## grupo
## g1 g2
##  1  3
table(grupo2)
## grupo2
## g1 g2 g3
##  1  3  0
```

Si no se usan factores no es posible especificar una cuenta de cero (como para el grupo 3), porque `table` no puede deducir la existencia de niveles sin ocurrencias.

La función `prop.table` produce las frecuencias relativas:

```
prop.table(table(grupo))
## grupo
##   g1   g2
## 0.25 0.75
prop.table(table(grupo2))
## grupo2
##   g1   g2   g3
## 0.25 0.75 0.00
```

Se puede consultar información relativa a los niveles de un factor:

```
nlevels(grupo2)  # cantidad de niveles
## [1] 3
levels(grupo2)   # niveles
## [1] "g1" "g2" "g3"
```

Internamente los factores se almacenan como un vector de enteros para ahorrar espacio:

```
typeof(grupo)
## [1] "character"
typeof(grupo2)
## [1] "integer"
as.integer(grupo2)
## [1] 1 2 2 2
```

La función `tapply` permite aplicar una función a un vector, agrupando sus datos por categorías. Por ejemplo, veamos la nota media del examen y la nota media por grupo:

```
mean(notas)
## [1] 7.75
tapply(notas, grupo2, mean)
## g1 g2 g3
## 10  7 NA
```

Si las categorías tienen un orden natural se puede usar la función `ordered` en lugar de `factor` para crear el factor:

```
nombres <- c("Ana", "Simón", "Nuria")
notas <- ordered(c("Notable", "Aprobado", "Sobresaliente"),
                 levels = c("Suspenso", "Aprobado", "Notable",
                            "Sobresaliente"))
notas
## [1] Notable       Aprobado      Sobresaliente
## Levels: Suspenso < Aprobado < Notable < Sobresaliente
```

Por lo demás, un factor ordenado se comporta igual que uno en el que no existe un orden de las categorías.

Para poder introducir un valor en un factor, este debe pertenecer a los niveles del factor:

```
notas[1] <- "Sobresaliente" # se modifica la nota
notas
## [1] Sobresaliente Aprobado      Sobresaliente
## Levels: Suspenso < Aprobado < Notable < Sobresaliente
notas[1] <- "Excelente"
## Warning in `[<-.factor`(`*tmp*`, 1, value = "Excelente"): invalid factor level,
## NA generated
notas
## [1] <NA>          Aprobado      Sobresaliente
## Levels: Suspenso < Aprobado < Notable < Sobresaliente
```

La función `cut` sirve para discretizar una variable real y convertirla en un factor. Vamos a ver un ejemplo (usa `?cut` para ver más posibilidades):

```
(notas <- runif(9, min = 0, max = 10))
## [1] 4.2967153 6.5165567 5.6773775 1.1350898 5.9592531 3.5804998 4.2880942
## [8] 0.5190332 2.6417767
n2 <- cut(notas, c(0, 5, 7, 9, 10))
```

```
n2
## [1] (0,5] (5,7] (5,7] (0,5] (5,7] (0,5] (0,5] (0,5] (0,5]
## Levels: (0,5] (5,7] (7,9] (9,10]
table(n2)
## n2
##  (0,5]  (5,7]  (7,9] (9,10]
##      6      3      0      0
n3 <- cut(notas,
          c(0, 5, 7, 9, 10),
          labels = c("Suspenso", "Aprobado", "Notable", "Sob"))
n3
## [1] Suspenso Aprobado Aprobado Suspenso Aprobado Suspenso Suspenso Suspenso
## [9] Suspenso
## Levels: Suspenso Aprobado Notable Sob
table(n3)
## n3
## Suspenso Aprobado  Notable      Sob
##        6        3        0        0
```

El primer parámetro de `cut` es un vector con los valores reales y el segundo un vector que define los intervalos usados para discretizar. Por ejemplo, si el segundo vector se llama `s`, por defecto, el primer intervalo será (s[0], s[1]], el segundo (s[1], s[2]] y así sucesivamente.

5.4 Números complejos

Aunque en estos apuntes no los vamos a utilizar, pues en estadística rara vez son necesarios, R permite trabajar con números complejos:

```
n <- -4 + 0i
sqrt(n)
## [1] 0+2i
```

5.5 Ejercicios

1. Crea un vector formado por 15 ocurrencias del valor 6.
2. Supón que quieres jugar al bingo. Escribe una expresión que sirva para simular la extracción de las 90 bolas de la urna.
3. Escribe una expresión que simule el lanzamiento de una moneda 10 veces. Obtén como resultado de la expresión un vector de valores `"cara"` y

`"cruz"`. Cuenta las ocurrencias de cara y cruz, y la proporción de cada valor.

4. La probabilidad de que una variable $X \sim \mathcal{N}(0, 1)$ sea mayor que 3 es `pnorm(3, lower.tail = FALSE)`. Genera un vector con un millón de instancias de una N(0, 1) y calcula qué proproción de ellas son mayores que 3.

5. Crea un vector aleatorio de 10 elementos y selecciona aquellos elementos del vector que ocupan índices impares. Usa `seq` para expresar el vector de índices impares.

6. Existen muchos métodos numéricos capaces de proporcionar aproximaciones a π. Usa la siguiente fórmula para calcular una aproximación (el número de términos de la sumatoria se leerá de teclado):

$$\pi = \sqrt{\sum_{i=1}^{\infty} \frac{6}{i^2}}$$

7. Dadas dos muestras emparejadas:

```
x <- iris$Petal.Length
y <- iris$Petal.Width
```

calcula su coeficiente de correlación lineal muestral de Pearson, según la siguiente fórmula, donde n es el tamaño de ambas muestras (puedes usar `sd`):

$$\frac{\sum_{i=1}^{n}(x_i - \overline{x})(y_i - \overline{y})}{(n-1)\sigma_x\sigma_y}$$

8. Crea un vector aleatorio con `sample(100, size = 10)`. Extrae aquellos elementos del vector que elevados al cuadrado sean menores que 625.

9. Escribe una expresión lógica que permita comprobar si un vector contiene exactamente los números del 1 al 20 sin repetidos (en cualquier orden).

10. Ejecuta:

```
v <- c(2, NA, 7)
all(v > 0)
## [1] NA
```

¿Tiene sentido la salida? Consulta la ayuda de `all` y escribe una expresión que compruebe si todos los elementos conocidos de un vector son mayores que cero.

11. Casi todas las operaciones con `NA` producen `NA`. Pero hay excepciones. Piensa en qué puede producir: `NA ^ 0`, `NA || TRUE` y `NA && FALSE`. Ejecuta las expresiones y comprueba si has acertado.

12. Compara la creación de las siguientes variables que almacena un valor categórico:

```
v1 <- c("Hombre", "Mujer", "Mujer", "hombre")
v2 <- factor(c("Hombre", "Mujer", "Mujer", "hombre"),
             levels = c("Mujer", "Hombre"))
```

    Ejecútalo y piensa por qué producen valores distintos.

13. El siguiente código guarda la longitud de pétalo y el tipo de 150 flores en los vectores `longitud` y `tipo`:

```
longitud <- iris$Petal.Length
tipo <- iris$Species
```

    El tipo es un factor. Calcula cuántas flores de cada tipo hay. Calcula la media de la longitud de pétalo por tipo de flor y la máxima longitud de pétalo para cada tipo de flor.

5.6 Soluciones

```
# Crear un vector formado por 15 ocurrencias de 6
rep(6, times = 15)
##  [1] 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6
```

```
# Bingo
sample(90)
##  [1] 24 13  8  7 27 82 29 84 50 26 33 87 81 85 30 68 51 54 59 32 11 80 42 75 89
## [26] 69 41 25 88 16 53 78 63 17 48 23 15 46 90 34  4 35 14 65 49 10 45 39 61 73
## [51] 31 44 37 38 58 18  3 22 43 71 76 86 36 64 21 66 40 56  2 77 20 74 57 67 55
## [76] 72  6  9 19 70 28 52  5 79  1 47 12 60 62 83
```

```
# Lanzamiento de una moneda 10 veces
(experimento <- sample(c("cruz", "cara"), size = 10, replace = TRUE))
##  [1] "cruz" "cara" "cara" "cruz" "cruz" "cara" "cruz" "cruz" "cara" "cara"
table(experimento)
## experimento
## cara cruz
##    5    5
```

```r
cat("Proporción caras:", mean(experimento == "cara"), "\n")
## Proporción caras: 0.5
cat("Proporción cruces:", mean(experimento == "cruz"))
## Proporción cruces: 0.5
```

```r
# Valores mayores que 3 de una N(0, 1)
pnorm(3, lower.tail = FALSE) # valor teórico
## [1] 0.001349898
v <- rnorm(1e6)
mean(v > 3)
## [1] 0.001308
```

```r
# Elementos que ocupan un índice impar
v <- sample(1:10, 10)
v
##  [1]  6  2  8  1  3  7  4  9 10  5
v[seq(1, length(v), by = 2)]
## [1]  6  8  3  4 10
```

```r
# Aproximación a pi
n <- as.integer(readline("Número de términos: "))
print(sqrt(sum(6 / (1:n) ^ 2)))
```

```r
# Coeficiente de correlación de Pearson
x <- iris$Petal.Length
y <- iris$Petal.Width
cor(x, y)                 # usando la función cor
## [1] 0.9628654
numerador   <- sum((x - mean(x)) * (y - mean(y)))
denominador <- (length(x) - 1) * sd(x) * sd(y)
numerador / denominador
## [1] 0.9628654
```

```r
# Elementos elevados al cuadrado menores que 625
(v <- sample(100, size = 10))
##  [1] 81 60 31 89  2 48 34 37 42 61
v[v * v < 625]
## [1] 2
```

```r
# Vector con valores del 1 al 20
sol = 1:20
v <- sample(1:20, size = 20)
length(v) == length(sol) && all(sort(v) == sol)
## [1] TRUE
```

```
v <- sample(1:20, size = 20, replace = TRUE)
length(v) == length(sol) && all(sort(v) == sol)
## [1] FALSE
v = 1:30
length(v) == length(sol) && all(sort(v) == sol)
## [1] FALSE
```

```
# Ejercicio sobre all
v = c(2, NA, 7)
all(v > 0)
## [1] NA
all(v > 0, na.rm = TRUE)
## [1] TRUE
```

```
# Expresiones con NA
NA ^ 0
## [1] 1
NA || TRUE
## [1] TRUE
NA && FALSE
## [1] FALSE
```

```
# Factor con valores Hombre y Mujer
(v1 <- c("Hombre", "Mujer", "Mujer", "hombre"))
## [1] "Hombre" "Mujer"  "Mujer"  "hombre"
(v2 <- factor(c("Hombre", "Mujer", "Mujer", "hombre"),
              levels = c("Mujer", "Hombre")))
## [1] Hombre Mujer  Mujer  <NA>
## Levels: Mujer Hombre
# Lo que ocurre es que nos hemos equivocado al teclear hombre con una h inicial en min
```

```
# Flores
longitud <- iris$Petal.Length
tipo <- iris$Species
table(tipo) # Cuenta ocurrencias de cada tipo de flor
## tipo
##     setosa versicolor  virginica
##         50         50         50
tapply(longitud, tipo, mean) # media de la longitud de pétalo por tipo
##     setosa versicolor  virginica
##      1.462      4.260      5.552
tapply(longitud, tipo, max) # máximo de la longitud de pétalo por tipo
##     setosa versicolor  virginica
##        1.9        5.1        6.9
```

Chapter 6

Tipos de datos: matrices

En este tema seguimos analizando los principales tipos de datos de R, estudiando las matrices. Las matrices son un tipo de dato que implementa el concepto matemático del mismo nombre. En R, una matriz es un vector que tiene asociado dos dimensiones. Todos los elementos de una matriz deben ser del mismo tipo. Podemos crear una matriz con la función `matrix`:

```
m <- matrix(c(2, 3, 6, 5), nrow = 2, ncol = 2)
m
##      [,1] [,2]
## [1,]    2    6
## [2,]    3    5
```

El primer parámetro es un vector e indica los elementos de la matriz. `nrow` sirve para especificar el número de filas y `ncol` el número de columnas. Si a partir de uno de los valores de `nrow` o `ncol` se puede deducir el otro, no es preciso especificar ambos:

```
matrix(c(2, 3, 6, 5), nrow = 2)
##      [,1] [,2]
## [1,]    2    6
## [2,]    3    5
matrix(c(2, 3, 6, 5), ncol = 2)
##      [,1] [,2]
## [1,]    2    6
## [2,]    3    5
```

Por defecto los elementos del vector se almacenan por columnas, pero podemos usar el parámetro `byrow` para que se guarden por filas:

```
matrix(1:20, nrow = 2)
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
## [1,]    1    3    5    7    9   11   13   15   17    19
## [2,]    2    4    6    8   10   12   14   16   18    20
matrix(1:20, nrow = 2, byrow = TRUE)
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
## [1,]    1    2    3    4    5    6    7    8    9    10
## [2,]   11   12   13   14   15   16   17   18   19    20
```

Si el vector que especifica los elementos tiene un tamaño inferior al de la matriz, los elementos del vector se reciclan. Esto es útil para crear una matriz en la que todos sus elementos tienen el mismo valor:

```
matrix(0, nrow = 3, ncol = 3)
##      [,1] [,2] [,3]
## [1,]    0    0    0
## [2,]    0    0    0
## [3,]    0    0    0
```

Podemos cambiar el atributo `dim` de un vector para convertirlo en una matriz:

```
(v <- c(2, 5, 1.1, 10)) # v es un vector
## [1]  2.0  5.0  1.1 10.0
dim(v) <- c(2, 2)      # cambia v de vector a matriz
v
##      [,1] [,2]
## [1,]    2  1.1
## [2,]    5 10.0
```

6.1 Consultar el tamaño de una matriz

La función `length` indica cuántos elementos tiene una matriz, la función `dim` cuántos elementos hay en cada dimensión, `nrow` el número de filas y `ncol` el número de columnas.

```
(m <- matrix(5, 2, 4))
##      [,1] [,2] [,3] [,4]
## [1,]    5    5    5    5
## [2,]    5    5    5    5
length(m)
## [1] 8
dim(m)     # dim devuelve un vector de tamaño 2
```

```
## [1] 2 4
nrow(m)
## [1] 2
ncol(m)
## [1] 4
```

6.2 Filas y columnas con nombres

Para algunas matrices puede resultar útil que las distintas filas y/o columnas tengan asociado un nombre. Los nombres se pueden especificar al crear la matriz con el parámetro `dimnames` de la función `matrix`. También de esta forma:

```
notas <- matrix(runif(9, 1, 10), nrow = 3)
rownames(notas) <- c("Juan", "María", "Jaime")
colnames(notas) <- paste("Examen", 1:3)
notas
##       Examen 1 Examen 2 Examen 3
## Juan  2.919869 8.121438 2.875558
## María 5.387534 1.406623 7.651563
## Jaime 8.443861 9.569262 1.582773
```

Los nombres se especifican mediante un vector de cadenas de caracteres. Es posible consultar los nombres de una dimensión o incluso anularlos:

```
rownames(notas)
## [1] "Juan"  "María" "Jaime"
colnames(notas)
## [1] "Examen 1" "Examen 2" "Examen 3"
colnames(notas) <- NULL # Anula los nombres de las columnas
notas
##           [,1]     [,2]     [,3]
## Juan  2.919869 8.121438 2.875558
## María 5.387534 1.406623 7.651563
## Jaime 8.443861 9.569262 1.582773
```

6.3 Aritmética de matrices

Al igual que ocurre con los vectores, los operadores aritméticos (`+`, `-`, `/`, `^`, …) son aplicables a matrices numéricas y trabajan elemento a elemento, reciclando si es necesario:

```
(m1 <- matrix(10, nrow = 2, ncol = 2))
##      [,1] [,2]
## [1,]   10   10
## [2,]   10   10
(m2 <- matrix(1:4, nrow = 2))
##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4
m1 - m2^2 + 3
##      [,1] [,2]
## [1,]   12    4
## [2,]    9   -3
(m3 <- matrix(rnorm(10), nrow = 2))
##           [,1]         [,2]       [,3]       [,4]       [,5]
## [1,] -2.954785  0.045731655  2.4111822 -0.6221762 -1.0570994
## [2,] -1.715616 -0.009431119 -0.7960108 -0.3924676  0.6630907
abs(m3) # aplica la función absoluto a todos los elementos de m3
##          [,1]        [,2]      [,3]      [,4]      [,5]
## [1,] 2.954785 0.045731655 2.4111822 0.6221762 1.0570994
## [2,] 1.715616 0.009431119 0.7960108 0.3924676 0.6630907
```

El producto clásico de matrices se realiza con el operador `%*%`:

```
(m1 <- matrix(1, nrow = 2, ncol = 2))
##      [,1] [,2]
## [1,]    1    1
## [2,]    1    1
(m2 <- matrix(1:4, nrow = 2))
##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4
m1 * m2   # multiplicación elemento a elemento
##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4
m1 %*% m2 # multiplicación clásica
##      [,1] [,2]
## [1,]    3    7
## [2,]    3    7
```

6.4 Funciones aplicables a matrices

Existen muchas funciones que aplican una operación al conjunto de los elementos de una matriz. Son tantas que no es posible enumerarlas todas, pero ten presente

que si quieres hacer un procesamiento típico con matrices es muy probable que esté implementado en alguna función de R.

Por ejemplo, `mean` calcula la media de los elementos de una matriz. También se puede calcular la media de las distintas columnas o filas con `colMeans` y `rowMeans` respectivamente.

```
m <- matrix(sample(4, size = 9, replace = TRUE), nrow = 3)
m
##      [,1] [,2] [,3]
## [1,]    3    1    1
## [2,]    3    2    2
## [3,]    2    4    4
mean(m)     # media de todos los elementos
## [1] 2.444444
rowMeans(m) # media de las filas
## [1] 1.666667 2.333333 3.333333
colMeans(m) # media de las columnas
## [1] 2.666667 2.333333 2.333333
```

`sum`, `colSums` y `rowSums` son análogas, pero suman los elementos de una matriz. Hay funciones, como `median`, que pueden aplicarse al conjunto formado por todos los elementos de una matriz, pero no existen versiones para aplicar el procesamiento por filas o columnas. En caso de que tengamos la necesidad de aplicar una función a las filas o columnas de una matriz se puede usar la función `apply`:

```
(m <- matrix(sample(4, size = 9, replace = TRUE), nrow = 3))
##      [,1] [,2] [,3]
## [1,]    3    3    3
## [2,]    4    4    4
## [3,]    1    3    2
rowSums(m)
## [1]  9 12  6
apply(m, 1, sum) # mismos efectos que rowSums
## [1]  9 12  6
colSums(m)
## [1]  8 10  9
apply(m, 2, sum) # mismos efectos que colSums
## [1]  8 10  9
```

La sintaxis de `apply` es `apply(matriz, dimensión, función)` y el efecto es aplicar la función indicada en el tercer parámetro a la matriz especificada en el primer parámetro en una dimensión. Si dimensión es 1 se aplica a las filas y si es 2 a las columnas. En caso de que la invocación a la función requiera

argumentos adicionales, estos pueden ser indicados después del argumento que especifica la función:

```
(m <- matrix(sample(c(1:5, NA), size = 12, replace = TRUE), nrow = 3))
##      [,1] [,2] [,3] [,4]
## [1,]    2   NA    2   NA
## [2,]   NA    1    2    5
## [3,]    5    3    1    3
apply(m, 2, median) # mediana por columnas
## [1] NA NA  2 NA
apply(m, 2, median, na.rm = TRUE)
## [1] 3.5 2.0 2.0 4.0
```

En la segunda llamada a `apply` el parámetro `na.rm` se le pasa a la función `median`.

Vamos a seguir describiendo algunas funciones de R que trabajan con matrices. `var` calcula la matriz de covarianzas:

```
m <- matrix(rnorm(1e4, mean = 0, sd = 2), nrow = 2500)
var(m)
##              [,1]        [,2]        [,3]        [,4]
## [1,]  4.15948356  0.07895521 -0.01427308  0.14811542
## [2,]  0.07895521  4.13455525  0.10596595 -0.01191684
## [3,] -0.01427308  0.10596595  3.90455761  0.10591173
## [4,]  0.14811542 -0.01191684  0.10591173  3.95010502
```

`var` considera cada columna de la matriz como una muestra. Observa la salida: en la diagonal aparece la varianza de cada muestra y en la posición (x, y) la covarianza entre las muestras de las columnas x e y. En el ejemplo las varianzas tienen un valor cercano a 4 (son muestras de una $N(0, 2)$) y las covarianzas son casi 0 (pues las muestras son independientes entre sí).

La función `t` calcula la traspuesta:

```
(m <- matrix(1:15, ncol = 5, byrow = TRUE))
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    2    3    4    5
## [2,]    6    7    8    9   10
## [3,]   11   12   13   14   15
t(m)
##      [,1] [,2] [,3]
## [1,]    1    6   11
## [2,]    2    7   12
## [3,]    3    8   13
## [4,]    4    9   14
## [5,]    5   10   15
```

La función `diag` extrae la diagonal de una matriz:

```
(m <- matrix(1:9, ncol = 3, byrow = TRUE))
##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]    4    5    6
## [3,]    7    8    9
diag(m)
## [1] 1 5 9
diag(m) <- 0
m
##      [,1] [,2] [,3]
## [1,]    0    2    3
## [2,]    4    0    6
## [3,]    7    8    0
```

La función `diag` aplicada a un vector crea una matriz con dicho vector como diagonal:

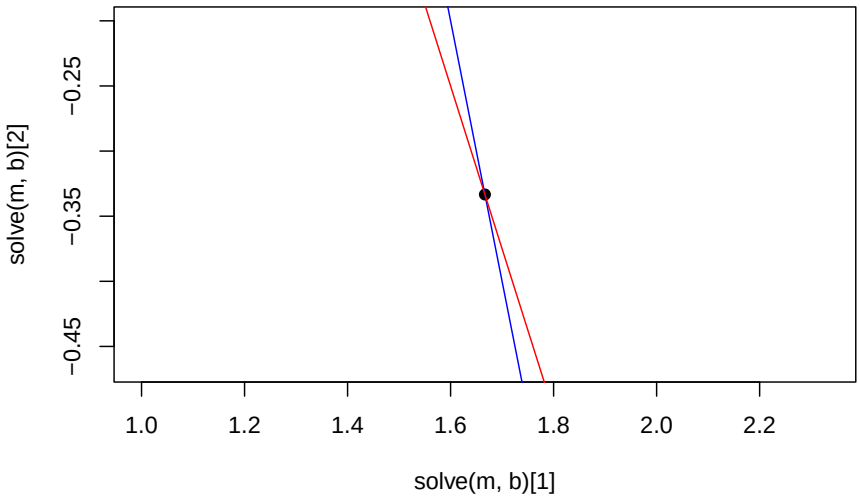
```
diag(3:1)
##      [,1] [,2] [,3]
## [1,]    3    0    0
## [2,]    0    2    0
## [3,]    0    0    1
```

Las funciones `det` y `determinant` calculan el determinante de una matriz. La función `solve` resuelve un sistema de ecuaciones lineales. Por ejemplo, suponga-amos que queremos saber dónde intersectan las rectas $2x + y - 3$ y $5x + 4y - 7$:

```
m <- matrix(c(2, 5, 1, 4), nrow = 2)
b <- c(3, 7)
solve(m, b)
## [1]  1.6666667 -0.3333333
```

Vamos a comprobar gráficamente que el cálculo es correcto:

```
plot(solve(m, b)[1], solve(m, b)[2], pch = 19)
abline(3, -2, col = "blue")
abline(7/4, -5/4, col = "red")
```



6.5 Matrices lógicas

Al igual que ocurre con los vectores, se puede aplicar operadores relacionales para crear matrices lógicas.

```
(m <- matrix(runif(8), nrow = 2))
##           [,1]      [,2]      [,3]      [,4]
## [1,] 0.7525039 0.6166801 0.5752071 0.5650909
## [2,] 0.2374461 0.3812454 0.8140243 0.2679709
m >= 0.75
##       [,1]  [,2]  [,3]  [,4]
## [1,]  TRUE FALSE FALSE FALSE
## [2,] FALSE FALSE  TRUE FALSE
```

Podemos contar y calcular la proporción de los elementos de una matriz que verifican una condición:

```
sum(m >= 0.75)  # ¿Cuántos elementos de m son >= 0.75?
## [1] 2
mean(m >= 0.75) # ¿Y la proporción?
## [1] 0.25
```

También podemos aplicar los operadores lógicos (`&`, `|` y `!`) a matrices lógicas, así como las funciones `any` y `all`, de la misma forma que se estudió con los vectores.

```
# Se simula lanzar 4 veces dos dados
(m <- matrix(sample(6, size = 8, replace = TRUE), nrow = 2))
##      [,1] [,2] [,3] [,4]
## [1,]    5    6    1    3
## [2,]    3    4    1    4
sum(m == 5 | m == 3) # ¿Cuántas veces sale el 3 o el 5?
## [1] 3
any(m == 6)          # ¿Ha salido el 6 al menos una vez?
## [1] TRUE
all(m != 1)          # ¿No ha salido el 1?
## [1] FALSE
```

6.6 Indexación de matrices

La indexación de matrices es similar a la de los vectores, pero es preciso indicar los índices de las dos dimensiones separados por una coma. Si alguna dimensión se deja vacía se seleccionan todos los elementos de dicha dimensión. Vamos a estudiar los 6 tipos de indexación que admiten las matrices. Los cuatro primeros son análogos a los usados en los vectores. Hay que tener en cuenta que con las matrices hay que especificar índices para las filas y para las columnas, con lo que se puede mezclar dos tipos de indexación, por ejemplo, indexar las filas con un vector lógico y las columnas con un vector numérico. En casi todos los ejemplos de esta sección las matrices son indexadas para consultar sus valores, pero también es posible indexar una matriz para modificarla.

6.6.1 Mediante un vector de enteros positivos

Se seleccionan los índices indicados en el vector:

```
(m <- matrix(1:16, nrow = 4))
##      [,1] [,2] [,3] [,4]
## [1,]    1    5    9   13
## [2,]    2    6   10   14
## [3,]    3    7   11   15
## [4,]    4    8   12   16
m[c(1, 3), 3:4]  # filas 1 y 3, columnas 3 y 4
##      [,1] [,2]
## [1,]    9   13
## [2,]   11   15
```

```
m[2, 2]          # fila 2, columna 2
## [1] 6
m[3:4, ]         # filas 3 y 4 (todas las columnas)
##      [,1] [,2] [,3] [,4]
## [1,]    3    7   11   15
## [2,]    4    8   12   16
```

6.6.2 Mediante un vector de enteros negativos

El vector indica los índices no seleccionados, en lugar de los seleccionados:

```
(m <- matrix(1:16, nrow = 4))
##      [,1] [,2] [,3] [,4]
## [1,]    1    5    9   13
## [2,]    2    6   10   14
## [3,]    3    7   11   15
## [4,]    4    8   12   16
m[, -1]  # todas las columnas, salvo la primera
##      [,1] [,2] [,3]
## [1,]    5    9   13
## [2,]    6   10   14
## [3,]    7   11   15
## [4,]    8   12   16
m[1:2, -c(1, ncol(m))] # filas 1 y 2. Todas las columnas salvo la primera y la última
##      [,1] [,2]
## [1,]    5    9
## [2,]    6   10
```

6.6.3 Mediante valores lógicos

Es muy útil, pues permite filtrar aquellas filas o columnas que verifiquen una condición. Por ejemplo:

```
(m <- matrix(1:16, nrow = 4))
##      [,1] [,2] [,3] [,4]
## [1,]    1    5    9   13
## [2,]    2    6   10   14
## [3,]    3    7   11   15
## [4,]    4    8   12   16
m[m[, 1] %% 2 == 0, ]
##      [,1] [,2] [,3] [,4]
## [1,]    2    6   10   14
## [2,]    4    8   12   16
```

Selecciona las filas en las que la columna 1 tiene un valor par. Veamos otro ejemplo con una condición un poco más compleja: vamos a seleccionar aquellas columnas de una matriz que no tienen valores `NA`.

```
(m <- matrix(sample(c(NA, 1:3), size = 12, replace = TRUE), nrow = 2))
##      [,1] [,2] [,3] [,4] [,5] [,6]
## [1,]    1   NA    2    3    2    2
## [2,]    3   NA    3    3    1    1
condicion <- apply(is.na(m), 2, any)
m[, !condicion]
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    2    3    2    2
## [2,]    3    3    3    1    1
```

También se puede usar una matriz lógica para indexar. Por ejemplo:

```
(m <- matrix(sample(c(1:3, NA), size = 12, replace = TRUE), nrow = 3))
##      [,1] [,2] [,3] [,4]
## [1,]   NA    2   NA    2
## [2,]    2    2    2    1
## [3,]    1   NA   NA    1
is.na(m) # produce una matriz lógica
##       [,1]  [,2]  [,3]  [,4]
## [1,]  TRUE FALSE  TRUE FALSE
## [2,] FALSE FALSE FALSE FALSE
## [3,] FALSE  TRUE  TRUE FALSE
m[is.na(m)] <- 0 # indexa con una matriz lógica
m
##      [,1] [,2] [,3] [,4]
## [1,]    0    2    0    2
## [2,]    2    2    2    1
## [3,]    1    0    0    1
```

6.6.4 Mediante un vector de cadenas de caracteres

Para usar esta posibilidad las filas y/o columnas deben tener nombres asociados:

```
notas <- matrix(runif(9, 1, 10), nrow = 3)
rownames(notas) <- c("Juan", "María", "Jaime")
colnames(notas) <- paste("Examen", 1:3)
notas
##       Examen 1 Examen 2 Examen 3
## Juan  9.572151 1.062372 8.270976
## María 2.346013 5.809828 4.928400
```

```
## Jaime 5.270132 9.441064 1.965953
notas[, "Examen 3"] # sólo el examen 3
##     Juan    María    Jaime
## 8.270976 4.928400 1.965953
notas[c("Juan", "Jaime"), ]
##       Examen 1 Examen 2 Examen 3
## Juan  9.572151 1.062372 8.270976
## Jaime 5.270132 9.441064 1.965953
```

Observa que en el primer ejemplo, al seleccionarse una única columna, R devuelve un vector en lugar de una matriz. Más adelante comentaremos cómo podemos hacer que R devuelva una matriz con una única columna.

6.6.5 Indexación lineal

Aunque no se use habitualmente, una matriz puede ser indexada con un único vector de índices. El resultado de esta indexación es un vector con los elementos de la matriz que ocupan los índices lineales indicados. Los índices lineales hacen referencia al vector asociado a la matriz, éste se puede obtener con la función `c`.

```
(m <- matrix(sample(10, 6), nrow = 2))
##      [,1] [,2] [,3]
## [1,]    2    5    9
## [2,]    8    6    7
c(m)  # vector asociado a m
## [1] 2 8 5 6 9 7
```

Como vemos, el vector asociado se construye concatenando las columnas de la matriz, y es a este vector al que se le aplica la indexación lineal:

```
m[c(1, 3, length(m))] # elementos 1, 3 y último de la versión lineal de m
## [1] 2 5 7
```

6.6.6 Mediante matriz de índices

Por último, una matriz se puede indexar con una matriz de dos columnas cuyas filas indican los distintos índices a indexar.

```
(m <- matrix(sample(100, 10), nrow = 2))
##      [,1] [,2] [,3] [,4] [,5]
## [1,]   13   27   34    4   72
## [2,]   45   75   47   55   46
```

```
i <- matrix(c(1, 1,
        1, 5,
        2, 3), ncol = 2, byrow = TRUE)
i # matriz con los índices a los que se quiere acceder
##      [,1] [,2]
## [1,]    1    1
## [2,]    1    5
## [3,]    2    3
m[i]
## [1] 13 72 47
```

6.7 Indexado que genera una única fila o columna

Si al indexar una matriz el resultado tiene una única fila o columna, R, por defecto, devuelve un vector con la fila o columna seleccionada, en lugar de devolver una matriz. Este comportamiento por defecto no es quizás lo más indicado. En el siguiente código se ejemplifica cómo cambiar el comportamiento por defecto, para poder obtener una matriz:

```
(m <- matrix(1:10, nrow = 2))
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    3    5    7    9
## [2,]    2    4    6    8   10
m[1, ] # fila 1, se obtiene un vector
## [1] 1 3 5 7 9
m[, 2] # columna 2, se obtiene un vector
## [1] 3 4
m[1, , drop = FALSE] # se obtiene una matriz
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    3    5    7    9
m[, 2, drop = FALSE] # se obtiene una matriz
##      [,1]
## [1,]    3
## [2,]    4
```

6.8 Concatenación de matrices

Se puede concatenar verticalmente matrices con el mismo número de columnas con `rbind`. Para concatenar horizontalmente matrices con el mismo número de filas se usa la función `cbind`:

```
(m1 <- matrix(c(1, 1, 2, 2), nrow = 2))
##      [,1] [,2]
## [1,]    1    2
## [2,]    1    2
(m2 <- m1 ^ 2)
##      [,1] [,2]
## [1,]    1    4
## [2,]    1    4
cbind(m1, m2)     # cbind concatena las columnas
##      [,1] [,2] [,3] [,4]
## [1,]    1    2    1    4
## [2,]    1    2    1    4
rbind(m1, m1 * 3) # rbind concatena las filas
##      [,1] [,2]
## [1,]    1    2
## [2,]    1    2
## [3,]    3    6
## [4,]    3    6
```

Las funciones `cbind` y `rbind` admiten el reciclado:

```
(m <- matrix(1:6, nrow = 3))
##      [,1] [,2]
## [1,]    1    4
## [2,]    2    5
## [3,]    3    6
cbind(m , 0) # añade una columna de ceros
##      [,1] [,2] [,3]
## [1,]    1    4    0
## [2,]    2    5    0
## [3,]    3    6    0
```

6.9 Arrays

Los vectores y matrices son un caso particular de *arrays*. Los vectores son *arrays* de una dimensión y las matrices de dos dimensiones. Dado que los *arrays* de 3 o más dimensiones se utilizan con poca frecuencia no los vamos a cubrir en estos apuntes. Sólo vamos a ver un ejemplo. Supongamos que queremos almacenar las 3 notas de n alumnos en 2 semestres. Esto se puede almacenar de una forma ordenada en un *array* tridimensional nx3x2:

```
n <- 5
notas <- array(runif(n*3*2, max = 10), dim = c(n, 3, 2))
```

```
notas
## , , 1
##
##           [,1]      [,2]      [,3]
## [1,] 9.125381 2.1338803 7.3797824
## [2,] 1.323580 1.7471532 7.9964874
## [3,] 7.165681 5.2841235 3.5013049
## [4,] 2.138362 0.2949213 1.7973755
## [5,] 7.571489 7.8237982 0.1531315
##
## , , 2
##
##            [,1]     [,2]     [,3]
## [1,] 3.5000069 7.969093 7.983221
## [2,] 4.7810746 5.343526 2.778460
## [3,] 5.3099247 1.408178 3.451532
## [4,] 9.3144338 2.891432 7.594464
## [5,] 0.7875821 8.449593 8.737541
notas[, , 1]  # notas del primer semestre
##           [,1]      [,2]      [,3]
## [1,] 9.125381 2.1338803 7.3797824
## [2,] 1.323580 1.7471532 7.9964874
## [3,] 7.165681 5.2841235 3.5013049
## [4,] 2.138362 0.2949213 1.7973755
## [5,] 7.571489 7.8237982 0.1531315
notas[1, , 2] # notas del alumno 1 en el segundo semestre
## [1] 3.500007 7.969093 7.983221
```

6.10 Ejercicios

1. Crea una matriz 5x5 cuya primera fila sean unos, la segunda doses y así sucesivamente.
2. Crea la matriz `matrix(sample(100, size = 12), nrow = 3)` y haz que todas las columnas aparezcan ordenadas.
3. Crea una matriz con `matrix(rnorm(15), nrow = 5)`. Selecciona las filas de la matriz cuya suma es positiva.
4. Haz un pequeño programa que solicite un entero y cree una matriz de tres columnas con su tabla de multiplicar. Por ejemplo, si se hubiera introducido el 3 se crearía la matriz (se muestran sólo las tres primeras filas):

   3 1 3
   3 2 6

3 3 9

5. Una matriz cuadrada es simétrica si es igual a su traspuesta. Escribe una expresión que dada una matriz `m` genere un valor lógico indicando si `m` es simétrica. Nota: puedes usar la función `all`.

6. Dada la matriz obtenida con `matrix(rnorm(10), nrow = 2)`, genera una expresión para calcular la cantidad de valores mayores que 0 en la matriz y otra para determinar si hay más valores positivos que negativos en la matriz.

7. Dada una matriz, añádele una columna con la suma de los elementos de cada fila de la matriz original.

8. Dada una matriz, obtén otra matriz formada por una permutación de sus fila.

Chapter 7

Tipos de datos: listas

En este tema seguimos analizando los principales tipos de datos de R, estudiando las listas. Una lista es similar a un vector, en el sentido de que puede almacenar una secuencia ordenada de elementos. La diferencia es que los elementos de un vector deben ser del mismo tipo, mientras que los de la lista pueden ser de distinto tipo. Esta diferencia hace que realmente sean dos tipos de datos relativamente distintos. Las listas son muy flexibles, pues permiten almacenar cualquier cosa. Los vectores son más específicos, lo que los hace más eficientes y permite que ciertas operaciones, como la suma o la aplicación de operaciones relacionales, tengan sentido.

Para crear una lista se usa la función `list`, especificando los elementos o *componentes* de la lista como argumentos de la función:

```
l <- list("Juan", 23, FALSE, c(8, 6.5, 9))
l
## [[1]]
## [1] "Juan"
##
## [[2]]
## [1] 23
##
## [[3]]
## [1] FALSE
##
## [[4]]
## [1] 8.0 6.5 9.0
```

Como ilustra el ejemplo, los componentes pueden ser de distinto tipo. Para algunas listas resulta más legible hacer que los distintos componentes tengan un

nombre. Esto se puede realizar directamente al crear la lista o, a posteriori, con la función `names`:

```
names(l) <- c("Nombre", "Edad", "Casado", "Notas") # añade nombres
l
## $Nombre
## [1] "Juan"
##
## $Edad
## [1] 23
##
## $Casado
## [1] FALSE
##
## $Notas
## [1] 8.0 6.5 9.0
punto2d <- list(x = 5, y = 6) # crea lista con nombres
punto2d
## $x
## [1] 5
##
## $y
## [1] 6
```

Dado que una lista puede contener componentes de distintos tipos (incluso de tipo lista), no resulta fácil visualizarla en la consola. La función `str` produce una visualización algo más estructurada en la consola:

```
str(l)
## List of 4
##  $ Nombre: chr "Juan"
##  $ Edad  : num 23
##  $ Casado: logi FALSE
##  $ Notas : num [1:3] 8 6.5 9
str(punto2d)
## List of 2
##  $ x: num 5
##  $ y: num 6
```

La función `str` es genérica, por lo que puede aplicarse a cualquier objeto, no sólo a listas. Otra función útil para visualizar el contenido de objetos es `View`, que abre un panel con información del objeto consultado en la zona del editor de texto de RStudio (pruébala).

Para obtener el número de componentes de una lista se usa la función `length`:

```
length(l)
## [1] 4
length(punto2d)
## [1] 2
```

7.1 Indexación

Existen varias formas de indexar una lista, así que usaremos la que nos guste más, nos parezca más legible o sea apropiada al cálculo que queramos realizar. Los operadores para indexar una lista son tres: `[]`, `[[]]` y `$`.

7.1.1 Indexación con `[]`

La indexación de listas con `[]` admite las mismas posibilidades que con vectores. Recordemos las posibilidades:

```
l[c(1, 4)]  # indexación con un vector de números positivos
## $Nombre
## [1] "Juan"
##
## $Notas
## [1] 8.0 6.5 9.0
l[-c(1, 4)] # indexación con un vector de números negativos
## $Edad
## [1] 23
##
## $Casado
## [1] FALSE
l[c("Nombre", "Edad")] # indexación con vector de cadenas
## $Nombre
## [1] "Juan"
##
## $Edad
## [1] 23
l[c(TRUE, FALSE, FALSE, TRUE)] # indexación con vector lógico
## $Nombre
## [1] "Juan"
##
## $Notas
## [1] 8.0 6.5 9.0
```

Al indexar con `[]` se obtiene una lista con los componentes seleccionados.

7.1.2 Indexación con [[]]

Es importante observar que al indexar con `[]` se obtiene una lista, aunque sólo accedamos a un componente de la lista. Por ejemplo, supongamos que queremos sumarle 2 al componente `x` del objeto `punto2d`:

```
punto2d <- list(x = 5, y = 6)
punto2d["x"]
## $x
## [1] 5
typeof(punto2d["x"])
## [1] "list"
# punto2d["x"] + 2 # pruébalo en la consola y obtendrás un error
```

No es posible realizar la suma con `[]`, porque no se puede aplicar el operador `+` a una lista. Cuando queramos **acceder al contenido de un único componente de una lista** para trabajar con él hay que usar el operador `[[]]` o el operador `$`:

```
punto2d[["x"]]
## [1] 5
typeof(punto2d[["x"]])
## [1] "double"
punto2d[["x"]] + 2
## [1] 7
```

Cuando usemos el operador `[[]]` habrá que utilizar un único índice:

```
l[[4]]          # accedemos al componente de índice 4
## [1] 8.0 6.5 9.0
l[["Notas"]]    # accedemos al componente de nombre Notas
## [1] 8.0 6.5 9.0
l[["Notas"]][1] # primera nota
## [1] 8
```

7.1.3 Indexación con $

Cuando una lista `l` tiene nombres asociados a sus componentes, las expresiones `l[["Nombre"]]` y `l$Nombre` son equivalentes:

```
l[["Notas"]]
## [1] 8.0 6.5 9.0
l$Notas
## [1] 8.0 6.5 9.0
```

Cuando tengamos un componente de una lista con un nombre muy largo, es posible especificarlo con el operador `$` escribiendo únicamente los tres primeros caracteres del nombre (salvo que haya ambigüedad porque más de un nombre de componente empiece por esos tres caracteres):

```
l$Not  # equivale a l$Notas
## [1] 8.0 6.5 9.0
```

El operador `$` es muy cómodo para acceder al contenido de un componente por su nombre.

7.2 Creación y eliminación de componentes

Es posible crear y eliminar componentes de una lista, veamos un ejemplo:

```
punto2d
## $x
## [1] 5
##
## $y
## [1] 6
punto2d$z <- 10   # se añade un componente de nombre z
punto2d
## $x
## [1] 5
##
## $y
## [1] 6
##
## $z
## [1] 10
punto2d$z <- NULL # se elimina el componente de nombre z
punto2d
## $x
## [1] 5
##
## $y
## [1] 6
```

A continuación se añade un componente usando indexación numérica:

```
punto2d[[3]] <- 9
punto2d
```

```
## $x
## [1] 5
##
## $y
## [1] 6
##
## [[3]]
## [1] 9
```

7.3 Las listas son un tipo de dato recursivo

Los componentes de una lista pueden ser de cualquier tipo, incluido el propio tipo lista. Un tipo de dato, como las listas, que admite elementos de su propio tipo se dice que es recursivo. Veamos un ejemplo de una lista con un componente de tipo lista:

```
fecha <- list(dia = 4, mes = "febrero", año = 2002)
alumno <- list(nombre = "Marta", nacimiento = fecha)
str(alumno)
## List of 2
##  $ nombre    : chr "Marta"
##  $ nacimiento:List of 3
##   ..$ dia: num 4
##   ..$ mes: chr "febrero"
##   ..$ año: num 2002
```

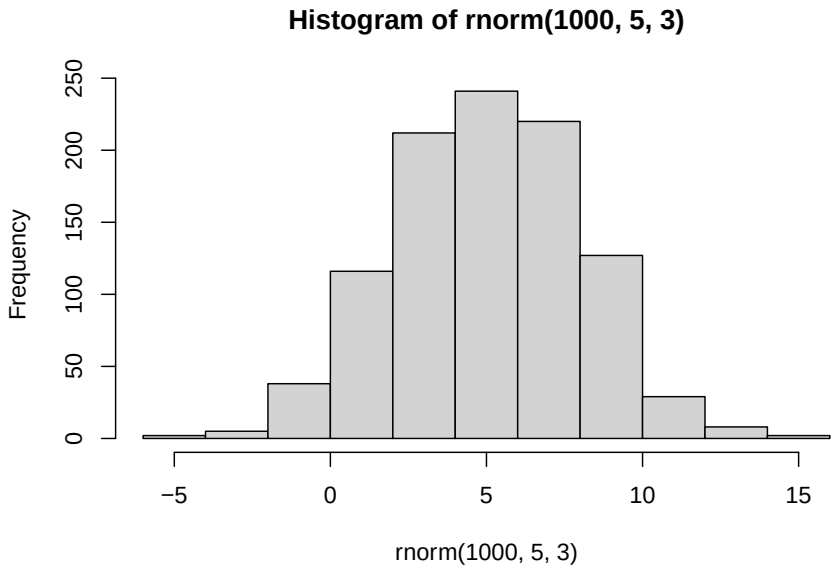
En este caso el componente `nacimiento` de la lista `alumno` es una lista. Veamos cómo acceder a sus campos:

```
alumno$nacimiento$año
## [1] 2002
alumno[["nacimiento"]][["año"]] # otra posibilidad
## [1] 2002
```

7.4 Funciones que devuelven listas

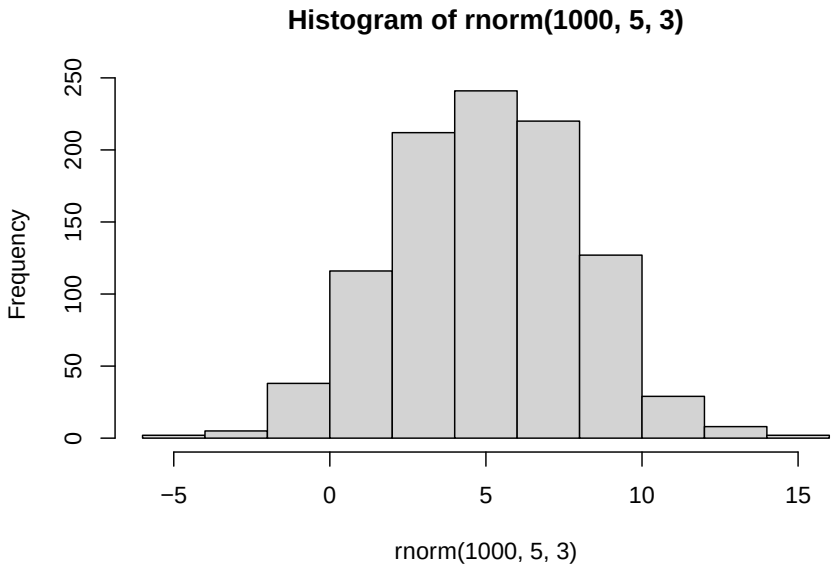
Muchas funciones de R devuelven información diversa sobre los cálculos que realizan. Como en R una función sólo puede devolver un objeto, la forma usual de devolver esta información es en una lista con distintos componentes, cada componente almacena algún tipo de dato sobre el cómputo realizado por la función. Veamos un ejemplo: la función `hist` calcula un histograma sobre una muestra y lo visualiza en la pantalla:

```
set.seed(10)
hist(rnorm(1000, 5, 3))
```



Pero `hist` no sólo visualiza el histograma, también devuelve una lista con información sobre sus características:

```
set.seed(10)
salida <- hist(rnorm(1000, 5, 3))
```




```
typeof(salida)
## [1] "list"
str(salida)
## List of 6
##  $ breaks  : num [1:12] -6 -4 -2 0 2 4 6 8 10 12 ...
##  $ counts  : int [1:11] 2 5 38 116 212 241 220 127 29 8 ...
##  $ density : num [1:11] 0.001 0.0025 0.019 0.058 0.106 ...
##  $ mids    : num [1:11] -5 -3 -1 1 3 5 7 9 11 13 ...
##  $ xname   : chr "rnorm(1000, 5, 3)"
##  $ equidist: logi TRUE
##  - attr(*, "class")= chr "histogram"
```

Los distintos componentes de la lista `salida` nos aportan información sobre el histograma construido. Así, el componente `breaks` nos indica que las cajas tienen rangos $[-6, -4]$, $[-4, -2]$, etc. El componente `equidist` nos indica que todas las cajas tienen la misma anchura. `counts` indica cuántos números caen en cada caja.

7.5 La función `lapply`

La función `lapply` permite aplicar una función a los distintos componentes de una lista. La salida de `lapply` es una lista. Por cada componente de la lista de

entrada se genera un componente en la lista de salida con el resultado de aplicar la función al componente de la lista de entrada:

```
str(l)
## List of 4
##  $ Nombre: chr "Juan"
##  $ Edad  : num 23
##  $ Casado: logi FALSE
##  $ Notas : num [1:3] 8 6.5 9
lapply(l, length) # longitud de los distintos componentes de l
## $Nombre
## [1] 1
##
## $Edad
## [1] 1
##
## $Casado
## [1] 1
##
## $Notas
## [1] 3
```

Si, como en este caso, la aplicación de la función produce elementos del mismo tipo (en concreto enteros), se puede usar la función `sapply` para obtener la salida en un vector:

```
sapply(l, length)
## Nombre   Edad Casado  Notas
##      1      1      1      3
```

La función `lapply` también se puede aplicar a vectores, en cuyo caso el vector se convierte a lista antes de aplicarle `lapply`:

```
v <- c(25, 4, 60)
lapply(v, sqrt)
## [[1]]
## [1] 5
##
## [[2]]
## [1] 2
##
## [[3]]
## [1] 7.745967
sapply(v, sqrt)
## [1] 5.000000 2.000000 7.745967
```

Para este ejemplo, sin embargo, sería más sencillo usar:

```
sqrt(v)
## [1] 5.000000 2.000000 7.745967
```

7.6 Listas multidimensionales

Por defecto las listas son unidimensionales, como los vectores. No obstante, se puede cambiar el atributo `dim` para convertir una lista unidimensional en multidimensional. En la práctica esta característica se usa muy pocas veces.

```
l <- list(1:3, "x", TRUE, 2.0)
dim(l) <-  c(2, 2)
l
##      [,1]      [,2]
## [1,] integer,3 TRUE
## [2,] "x"       2
l[[1, 1]]
## [1] 1 2 3
```

7.7 Ejercicios

1. Dada la lista: `list(rbinom(10, 30, 0.5), runif(100, -8, -4))` usa `lapply` o `sapply` para calcular las medias de los componentes de la lista.

Chapter 8

Tipos de datos: data frames

En este tema seguimos analizando los principales tipos de datos de R, estudiando los *data frames*. Un *data frame* es una estructura o tipo de dato muy usado en estadística, similar a una hoja de cálculo. Un *data frame* es parecido a una matriz, en el sentido de que es una estructura bidimensional con sus filas y columnas, pero con la diferencia de que las distintas columnas de un *data frame* pueden ser de distintos tipos. Esto lo hace muy versátil. Normalmente un *data frame* contienen distintas variables en las columnas que indican los valores asociados a cada fila. Cada fila constituye una observación. Si realizas análisis de datos en R vas a trabajar con *data frames*.

Internamente un *data frame* se implementa como una lista, cada componente de la lista es una columna del *data frame*. Todas las columnas deben tener la misma longitud. Las distintas columnas de un *data frame* serán vectores, aunque técnicamente es posible que sean matrices o listas.

El hecho de que un *data frame* sea semánticamente similar a una matriz, pero esté implementado como una lista, va a producir una gran riqueza sintáctica a la hora de acceder a sus elementos.

8.1 Creación de *data frames*

Normalmente los *data frames* se construirán a partir de datos que están en archivos tipo hoja de cálculo almacenados en un dispositivo de almacenamiento local (como un disco duro) o internet con funciones como `read.table`. También existen muchos paquetes que incluyen *data frames* entre sus datos. Por ejemplo:

```
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
```

En este caso se ha mostrado el famoso conjunto de datos sobre lirios (`iris`). Se puede obtener ayuda sobre los *data frames* incluidos en paquetes: prueba `?iris` para obtener ayuda sobre este conjunto de datos. La función `head` hace que se visualicen las primeras filas u observaciones del conjunto de datos. `head` es una función genérica y se puede aplicar a otros tipos de datos como vectores. Este *data frame* tiene 150 observaciones, con 5 variables por observación: longitud y anchura de sépalo, longitud y anchura de pétalo (todas en centímetros) y especie.

También se puede crear un *data frame* con la función `data.frame`, aportando los datos oportunos. La sintaxis de creación de un *data frame* es muy parecida a la de creación de una lista:

```
d <- data.frame(Nombre = c("Laura", "Mario", "Luis"),
                Edad = c(22, 43, 55),
                Ciudad = factor(c("Jaén", "Granada", "Jaén")),
                Casado = c(FALSE, TRUE, TRUE)
)
d
##   Nombre Edad  Ciudad Casado
## 1  Laura   22    Jaén  FALSE
## 2  Mario   43 Granada   TRUE
## 3   Luis   55    Jaén   TRUE
```

Observa que cada columna del *data frame* se ha especificado como un vector o factor, precedida por su nombre y el carácter `=`. Todas las columnas deben tener la misma longitud.

También es posible crear un *data frame* con la función `edit`, que permite editar de una forma sencilla los contenidos:

```
datos <- edit(data.frame()) # Abre un editor para introducir los datos
```

Una función parecida a `edit` es `View`, que nos permite visualizar los elementos de un *data frame* o matriz de manera similar a una hoja de cálculos:

```
View(iris)
```

8.2 Acceso a los elementos con formato matriz

Se puede acceder a los datos de un *data frame* como se hace con una matriz, es decir, especificando las filas y columnas a las que queremos acceder con la sintaxis: `df[filas, columnas]`. En concreto, se puede especificar las distintas filas y columnas usando vectores de:

- índices positivos
- índices negativos
- cadenas de caracteres
- valores lógicos

Vamos a ver algunos ejemplos:

```
d[c(1, 3), ]             # Índices positivos: filas 1 y 3
##   Nombre Edad Ciudad Casado
## 1  Laura   22   Jaén  FALSE
## 3   Luis   55   Jaén   TRUE
d[, -2]                  # Índices negativos: excluir columna 2
##   Nombre  Ciudad Casado
## 1  Laura    Jaén  FALSE
## 2  Mario Granada   TRUE
## 3   Luis    Jaén   TRUE
d[, c("Nombre", "Edad")] # cadenas de caracteres
##   Nombre Edad
## 1  Laura   22
## 2  Mario   43
## 3   Luis   55
d[d[, "Ciudad"] == "Jaén", ] # valores lógicos
##   Nombre Edad Ciudad Casado
## 1  Laura   22   Jaén  FALSE
## 3   Luis   55   Jaén   TRUE
```

Observa que la última operación filtra las observaciones (filas) cuya ciudad es Jaén. Si sólo se selecciona una columna de un *data frame*, el comportamiento por defecto es devolver un vector con los datos de dicha columna, en lugar de un *data frame* con una única columna. Veamos cómo anular el comportamiento por defecto:

```
d[, "Nombre"] # Selecciona 1 columna, devuelve un vector
## [1] "Laura" "Mario" "Luis"
d[, "Nombre", drop = FALSE] # Ahora devuelve un data frame
##   Nombre
## 1  Laura
```

```
## 2  Mario
## 3   Luis
```

8.3 Acceso a los elementos con formato lista

Como hemos comentado, un *data frame* se implementa como una lista cuyos componentes son las columnas del *data frame*. Esto permite acceder con formato de lista para seleccionar las columnas de un *data frame*. Se recuerda que los operadores de acceso a lista son: `[]`, `[[]]` y `$`.

Con `[]` podemos seleccionar una o varias columnas y el resultado será un *data frame* con las columnas seleccionadas:

```
d[1:3]     # primeras 3 columnas
##   Nombre Edad  Ciudad
## 1  Laura   22    Jaén
## 2  Mario   43 Granada
## 3   Luis   55    Jaén
d[c("Nombre", "Casado")] # columnas de nombres: Nombre y Casado
##   Nombre Casado
## 1  Laura  FALSE
## 2  Mario   TRUE
## 3   Luis   TRUE
d[ncol(d)] # última columna (produce un data frame)
##   Casado
## 1  FALSE
## 2   TRUE
## 3   TRUE
d[-ncol(d)] # Todas las columnas, salvo la última
##   Nombre Edad  Ciudad
## 1  Laura   22    Jaén
## 2  Mario   43 Granada
## 3   Luis   55    Jaén
```

Observa que al acceder con `[]` siempre se obtiene un *data frame*, aunque sólo se seleccione una columna. Si se accede con `[[]]` sólo se puede obtener una columna; además, se obtiene un vector con los contenidos de dicha columna, Por ejemplo, no tiene sentido `mean(d["Edad"])` (pues no tiene sentido aplicar `mean` a un *data frame*), pero sí `mean(d[["Edad"]])`.

```
d[["Nombre"]]     # Vector asociado a la columna Nombre
## [1] "Laura" "Mario" "Luis"
mean(d[[2]])      # Media de la columna 2
## [1] 40
```

```
mean(d[["Edad"]]) # Media de la columna Edad
## [1] 40
```

Por último, el operador `$` se utiliza para acceder al contenido de una columna mediante su nombre. `d$Edad` equivale a `d[["Edad"]]`, pero resulta algo más corto de escribir.

```
d$Ciudad
## [1] Jaén    Granada Jaén
## Levels: Granada Jaén
d[d$Casado, ] # Muestra sólo las personas casadas
##   Nombre Edad  Ciudad Casado
## 2  Mario   43 Granada   TRUE
## 3   Luis   55    Jaén   TRUE
```

8.4 Trabajando con *data frames*

Las posibilidades de procesamiento de un *data frame* son enormes. En general, podemos aplicar la mayor parte de las funciones asociadas a matrices y a listas. En este apartado vamos a ver algunas, pero ten presente que casi cualquier cálculo que quieras realizar sobre un *data frame* se puede realizar de manera elegante con unas pocas sentencias.

8.4.1 Consulta de información y nombres

Las funciones `length` y `ncol` devuelven el número de columnas de un *data frame*, mientras que `nrow` devuelve el número de filas. Todos los *data frames* tienen nombres de columnas y filas (por defecto los nombres de las filas son el número de observación), éstos se pueden consultar y modificar con `colnames` y `rownames`:

```
length(d)   # número de columnas
## [1] 4
ncol(d)     # número de columnas
## [1] 4
nrow(d)     # número de filas
## [1] 3
colnames(d) # nombres de las columnas
## [1] "Nombre" "Edad"   "Ciudad" "Casado"
names(d)    # también produce el nombre de las columnas
## [1] "Nombre" "Edad"   "Ciudad" "Casado"
rownames(d) # nombres de las filas
```

```
## [1] "1" "2" "3"
rownames(d) <- c("Primero", "Segundo", "Tercero")
d
##         Nombre Edad  Ciudad Casado
## Primero  Laura   22    Jaén  FALSE
## Segundo  Mario   43 Granada   TRUE
## Tercero   Luis   55    Jaén   TRUE
```

Las funciones `str` y `summary` (funciones genéricas aplicables a distintos objetos, no solo a *data frames*) nos dan información sobre un *data frame*:

```
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 .
summary(iris)
##   Sepal.Length    Sepal.Width     Petal.Length    Petal.Width
##  Min.   :4.300   Min.   :2.000   Min.   :1.000   Min.   :0.100
##  1st Qu.:5.100   1st Qu.:2.800   1st Qu.:1.600   1st Qu.:0.300
##  Median :5.800   Median :3.000   Median :4.350   Median :1.300
##  Mean   :5.843   Mean   :3.057   Mean   :3.758   Mean   :1.199
##  3rd Qu.:6.400   3rd Qu.:3.300   3rd Qu.:5.100   3rd Qu.:1.800
##  Max.   :7.900   Max.   :4.400   Max.   :6.900   Max.   :2.500
##        Species
##  setosa    :50
##  versicolor:50
##  virginica :50
##
##
##
```

`str` nos indica cuántas observaciones (filas) y variables (columnas) tiene el *data frame* y el nombre, tipo y varios valores de cada variable. `summary` nos ofrece información descriptiva de las distintas columnas. Si la columna es numérica nos ofrece el mínimo, máximo, cuartiles 1, 2 y 3 y media. Si la columna es un factor nos cuenta el número de ocurrencias de cada nivel.

8.4.2 Modificación de un *data frame*

Se puede modificar datos de un *data frame*:

```
d
##         Nombre Edad  Ciudad Casado
## Primero  Laura   22    Jaén  FALSE
## Segundo  Mario   43 Granada   TRUE
## Tercero   Luis   55    Jaén   TRUE
d[3, 3] <- "Granada" # d[3, "Ciudad] <- "Granada"
d
##         Nombre Edad  Ciudad Casado
## Primero  Laura   22    Jaén  FALSE
## Segundo  Mario   43 Granada   TRUE
## Tercero   Luis   55 Granada   TRUE
```

Es posible también añadir o eliminar nuevas filas y columnas de diferentes formas. Veamos algunos ejemplos:

```
d$Peso <- c(55.2, 72.1, 85) # se añade una columna
d
##         Nombre Edad  Ciudad Casado Peso
## Primero  Laura   22    Jaén  FALSE 55.2
## Segundo  Mario   43 Granada   TRUE 72.1
## Tercero   Luis   55 Granada   TRUE 85.0
d$Casado <- NULL            # se elimina una columna
d
##         Nombre Edad  Ciudad Peso
## Primero  Laura   22    Jaén 55.2
## Segundo  Mario   43 Granada 72.1
## Tercero   Luis   55 Granada 85.0
t <- data.frame(Nombre = "Marta", Edad = 18, Ciudad = "Jaén", Peso = 67)
d <- rbind(d, t) # se añade una fila
d
##         Nombre Edad  Ciudad Peso
## Primero  Laura   22    Jaén 55.2
## Segundo  Mario   43 Granada 72.1
## Tercero   Luis   55 Granada 85.0
## 1         Marta   18    Jaén 67.0
d2 <- d[3:nrow(d), ]            # se eliminan las dos primeras filas
d2
##         Nombre Edad  Ciudad Peso
## Tercero   Luis   55 Granada   85
## 1         Marta   18    Jaén   67
```

Las funciones `cbind` y `rbind` permiten concatenar *data frames* horizontal y verticalmente de manera respectiva.

8.4.3 Funciones apply

Dada la dicotomía lista-matriz de los *data frames* es posible aplicarles distintas variedades de la familia de funciones *apply*. Veamos algunos ejemplos:

```
apply(iris[, -5], 2, mean) # media por columnas de las 4 primeras columnas
## Sepal.Length  Sepal.Width Petal.Length  Petal.Width
##     5.843333     3.057333     3.758000     1.199333
lapply(iris[-5], mean)     # lapply aplica la función a las columnas
## $Sepal.Length
## [1] 5.843333
##
## $Sepal.Width
## [1] 3.057333
##
## $Petal.Length
## [1] 3.758
##
## $Petal.Width
## [1] 1.199333
sapply(iris[, -5], mean)
## Sepal.Length  Sepal.Width Petal.Length  Petal.Width
##     5.843333     3.057333     3.758000     1.199333
colMeans(iris[-5])         # también se puede usar colMeans
## Sepal.Length  Sepal.Width Petal.Length  Petal.Width
##     5.843333     3.057333     3.758000     1.199333
```

Observa que las funciones se aplican sobre `iris[, -5]`. Es decir, se elimina la última columna porque esta es un factor y no tiene sentido calcular su media. En este ejemplo, todos los cálculos producen el mismo resultado, aunque `lapply` genera como salida una lista en lugar de un vector.

También tiene sentido aplicar `tapply` usando las columnas de un *data frame*. Se recuerda que `tapply` aplica una función a los subgrupos o categorías de un factor. Por ejemplo, podemos calcular la media de las edades por ciudades:

```
d
##         Nombre Edad  Ciudad Peso
## Primero  Laura   22    Jaén 55.2
## Segundo  Mario   43 Granada 72.1
## Tercero   Luis   55 Granada 85.0
## 1        Marta   18    Jaén 67.0
tapply(d$Edad, d$Ciudad, mean) # media de edades por ciudad
## Granada    Jaén
##      49      20
tapply(d$Edad, d$Ciudad, max)  # máxima edad por ciudad
```

```
## Granada    Jaén
##      55      22
```

8.4.4 Casos completos

Un *data frame* puede contener valores perdidos para una o varias variables. La función `complete.cases` genera un vector lógico que indica si las distintas filas de un *data frame* no contienen valores perdidos, es decir, son observaciones completas.

```
datos <- data.frame(altura = rnorm(5, 1.7, 0.1),
                    peso = rnorm(5, 75, 10))
datos
##     altura     peso
## 1 1.805001 77.88344
## 2 1.728609 71.59608
## 3 1.724056 85.61335
## 4 1.783271 62.90951
## 5 1.677702 85.52407
datos[sample(nrow(datos), size = 1), "altura"] <- NA
datos[sample(nrow(datos), size = 1), "peso"] <- NA
datos
##     altura     peso
## 1 1.805001       NA
## 2 1.728609 71.59608
## 3       NA 85.61335
## 4 1.783271 62.90951
## 5 1.677702 85.52407
complete.cases(datos)
## [1] FALSE  TRUE FALSE  TRUE  TRUE
datos[complete.cases(datos), ]
##     altura     peso
## 2 1.728609 71.59608
## 4 1.783271 62.90951
## 5 1.677702 85.52407
```

8.4.5 Muestra aleatoria de un *data frame*

Se puede obtener una muestra aleatoria de las observaciones de un *data frame* usando `sample`. Veamos algunos ejemplos (ejecuta antes `?letters` para ver qué contiene este vector):

```
(df <- data.frame(a = 1:7, b = letters[1:7]))
##   a b
## 1 1 a
## 2 2 b
## 3 3 c
## 4 4 d
## 5 5 e
## 6 6 f
## 7 7 g
df[sample(nrow(df)), ]    # Permutación de las filas de df
##   a b
## 6 6 f
## 1 1 a
## 7 7 g
## 5 5 e
## 2 2 b
## 3 3 c
## 4 4 d
df[sample(nrow(df), 3), ] # Muestra de 3 filas
##   a b
## 3 3 c
## 7 7 g
## 4 4 d
# Muestra de 5 filas con reemplazo
df[sample(nrow(df), 5, replace = TRUE), ]
##     a b
## 7   7 g
## 2   2 b
## 3   3 c
## 6   6 f
## 3.1 3 c
```

8.4.6 Ordenación

Cuando estudiamos los vectores ya vimos la función `sort`, que permite ordenar un vector. Sin embargo, `sort` no nos sirve para ordenar las filas de un *data frame*:

```
set.seed(15)
df <- data.frame(a = 1:5, b = letters[5:1])
(df <- df[sample(nrow(df)), ])
##   a b
## 5 5 a
## 3 3 c
```

```
## 2 2 d
## 4 4 b
## 1 1 e
(df2 <- data.frame(a = sort(df$a), b = df$b))
##   a b
## 1 1 a
## 2 2 c
## 3 3 d
## 4 4 b
## 5 5 e
```

El problema es que se ordena la primera columna, pero la segunda queda como estaba y, por lo tanto, los datos de las distintas columnas están mezclados. Para ordenar por el contenido de una columna hay que usar la función `order`. Vamos a ver su funcionamiento con un ejemplo:

```
v <- c(23, 19, 5, 30, 18)
(o <- order(v))
## [1] 3 5 2 1 4
v[order(v)]
## [1]  5 18 19 23 30
```

La invocación `o <-order(v)` produce un vector (`o`) de índices de `v`, de forma que `o[1]` es el índice del menor elemento de `v`, `o[2]` es el índice del segundo elemento menor de `v` y así sucesivamente. Por lo tanto `v[o]` equivale a `sort[v]`. La ventaja es que `order` sirve para ordenar las filas de un *data frame* por el valor de cualquiera de sus columnas:

```
df[order(df$a), ] # ordenamos por la columna a
##   a b
## 1 1 e
## 2 2 d
## 3 3 c
## 4 4 b
## 5 5 a
df[order(df$b), ] # ordenamos por la columna b
##   a b
## 5 5 a
## 4 4 b
## 3 3 c
## 2 2 d
## 1 1 e
```

Además, `order` permite ordenar ascendente o descendentemente, así como por varios campos (por ejemplo, por apellido y nombre).

8.5 Ejercicios

1. Calcula cuántas flores de cada especie contiene el *data frame* iris (puedes usar la función `table`).
2. Calcula la desviación estándar de la anchura de pétalo para aquellas flores de la especie setosa del conjunto de datos `iris`.
3. Calcula la desviación estándar de la anchura de pétalo de las distintas especies del conjunto de datos `iris`.
4. Calcula la desviación estándar de todas las columnas numéricas de `iris` para aquellas flores de la especie setosa.
5. El *data frame* `mtcars` contiene información sobre varios coches (usa `?mtcars` y `head(mtcars)` para familiarizarte con él). El campo `mpg` indica cuántas millas puede recorrer un coche con un galón de combustible. Por lo tanto, cuanto mayor sea este campo más económico es el coche. Usa `range` para obtener el rango de valores de la columna `mpg`, selecciona también aquellas observaciones en las que `mpg` es igual o superior a 30.
6. Ordena el *data frame* `mtcar` en orden descendente por el campo `mgp`.
7. Para el conjunto de datos `mtcar`, ¿cuántos coches tiene 4 o 6 cilindros (columna `cyl`)?
8. ¿Por qué `mtcars[1:20]` produce un error y `mtcars[1:20, ]` no?
9. Genera un *data frame* a partir de `iris`, pero con sus columnas permutadas aleatoriamente.
10. Genera un *data frame* a partir de `iris`, pero con sus columnas ordenadas alfabéticamente (según el nombre de las columnas).
11. Arregla los siguientes errores comunes al indexar un *data frame*:

```
mtcars[mtcars$cyl = 4, ]
mtcars[mtcars$cyl <= 5]
mtcars[mtcars$cyl == 4 | 6, ]
```

Chapter 9

Estructuras condicionales

En este tema vamos a estudiar las instrucciones condicionales. Estas permiten ejecutar un código u otro en función del resultado de evaluar una condición lógica.

9.1 Sentencia `if`

La instrucción `if` permite ejecutar un conjunto de instrucciones, u otro, en función de una condición. Veamos un ejemplo:

```
dividendo <- 7
divisor <- 2
if (divisor != 0) {
  cat(dividendo, "/", divisor, "=", dividendo / divisor, "\n")
} else {
  print("No es posible dividir entre 0")
}
## 7 / 2 = 3.5
```

Observa la sintaxis. La sentencia comienza con la palabra reservada `if`, seguida de la expresión lógica encerrada entre paréntesis. Después, encerrada entre llaves (`{}`), viene el conjunto de instrucciones a ejecutar si el resultado de evaluar la expresión lógica es verdadero. La parte `else` (en otro caso) incluye otro conjunto de instrucciones (encerradas entre llaves) que se ejecutan si la expresión lógica es falsa. Modifica el código previo para que `divisor`valga 0 y comprueba cómo se ejecutan las instrucciones asociadas a la parte `else`. Los conjuntos de instrucciones se escriben indentados (es decir, desplazados) a la derecha para facilitar la lectura de la instrucción, aunque sintácticamente no es obligatorio indentar estos conjuntos de instrucciones.

La parte `else` es opcional, por lo que si no necesitamos ejecutar instrucciones si no se verifica la condición lógica podemos omitirla.

```
# Convierte la variable x a positiva
(x <- rnorm(1))
## [1] 2.285533
if (x < 0) {
  print("Cambio el signo")
  x <- -x
}
x
## [1] 2.285533
```

Si no existe parte `else` y sólo queremos ejecutar una instrucción, no es necesario usar las llaves para delimitar las instrucciones:

```
# Convierte la variable x a positiva
(x <- rnorm(1))
## [1] -0.6620538
if (x < 0) x <- -x
x
## [1] 0.6620538
```

La condición lógica debe producir un valor escalar de tipo lógico. No tiene sentido que la condición lógica sea un vector de valores lógicos con más de un elemento, aunque si ocurre esto se usa el primer elemento del vector como condición lógica.

```
v <- c(3, -2, -4)
if (v > 0) {
  print('El primer elemento de v es mayor que cero')
}
## Warning in if (v > 0) {: la condición tiene longitud > 1 y sólo el primer
## elemento será usado
## [1] "El primer elemento de v es mayor que cero"
```

Observa cómo se genera un aviso para indicar que la condición lógica es un vector y no un escalar.

9.1.1 Sentencias `if` anidadas

Las instrucciones asociadas a una sentencia `if` pueden ser de cualquier tipo, incluido la propia instrucción `if`. Vamos a ver un ejemplo con un código que calcula el máximo de tres valores:

```
x <- 7; y <- 8; z <- 6
if (x > y) {
  if (x > z) {
    mayor <- x
  } else {
    mayor <- z
  }
} else {
  if (y > z) {
    mayor <- y
  } else {
    mayor <- z
  }
}
mayor
## [1] 8
```

9.1.2 Selección múltiple con la sentencia `if`

El siguiente fragmento de código, que convierte una calificación numérica en una categórica, tiene la estructura lógica de una selección múltiple, en el sentido de que existen n conjuntos de instrucciones mutuamente excluyentes. Es decir, se dispone de n conjuntos de instrucciones de los que sólo se ejecutará (seleccionará) un conjunto.

```
(nota <- runif(1, min = 0, max = 10))
## [1] 8.31429
if (nota >= 9) {
  notac <- "Sobresaliente"
} else {
  if (nota >= 7) {
    notac <- "Notable"
  } else {
    if (nota >= 5) {
      notac <- "Aprobado"
    } else {
      notac <- "Suspenso"
    }
  }
}
notac
## [1] "Notable"
```

Para implementar una estructura lógica de este tipo el cuerpo de la parte `else` de todas las instrucciones `if` viene constituido por una única instrucción `if`,

lo que implica que no es necesario utilizar llaves para delimitar el cuerpo de la parte `else`. Esto posibilita el escribir la selección múltiple utilizando un formato alternativo como el mostrado a continuación.

```
(nota <- runif(1, min = 0, max = 10))
## [1] 1.046694
if (nota >= 9) {
  notac <- "Sobresaliente"
} else if (nota >= 7) {
  notac <- "Notable"
} else if (nota >= 5) {
  notac <- "Aprobado"
} else {
  notac <- "Suspenso"
}
notac
## [1] "Suspenso"
```

Este formato resulta más legible para muchas personas porque todas las expresiones lógicas y conjuntos de instrucciones aparecen al mismo nivel de indentación, reforzando la idea de que es una selección múltiple excluyente, es decir, en la que sólo se ejecuta un conjunto de instrucciones.

9.2 La función `ifelse`

La función `ifelse` es una versión vectorizada de la instrucción `if`. Tiene tres parámetros: un vector de valores lógicos y dos vectores de expresiones. Si los vectores de expresiones tienen menos elementos que el vector lógico, se reciclan a la longitud del vector lógico. Veamos un ejemplo:

```
v <- c(4, -2, 16, 25, -7)
ifelse(v > 0, v, -v) # equivale a abs(v)
## [1]  4  2 16 25  7
```

La función `ifelse` produce un vector del mismo tamaño que el vector de valores lógicos usado como primer parámetro. El funcionamiento es el siguiente: para los índices del vector lógico (primer argumento) con valores de verdadero se usa el elemento del mismo índice del primer vector de expresiones (en el ejemplo `v`). En caso de que el valor sea falso se usa el elemento del mismo índice del segundo vector de expresiones (en el ejemplo `-v`). Veamos un ejemplo en el que se recicla el segundo vector de expresiones:

```
v <- c(4, -2, 16, 25, -7)
ifelse(v > 0, v, 0)
## [1]  4  0 16 25  0
ifelse(v > 0, v, rep(0, length(v))) # equivale al anterior
## [1]  4  0 16 25  0
```

9.3 La sentencia `switch` (avanzado)

Esta sección es avanzada y su contenido no es evaluable.

La sentencia `if` produce un resultado al ser ejecutada: el resultado de evaluar la última instrucción que ejecuta de sus conjuntos de instrucciones asociados. Por ejemplo, supongamos que queremos asignar a la variable signo el valor `"positivo"` o `"negativo"` en función del signo de una variable. Esto lo podemos implementar así:

```
(x <- rnorm(1))
## [1] 0.3749493
if (x >= 0) {
  signo <- "positivo"
} else {
  signo <- "negativo"
}
signo
## [1] "positivo"
```

Pero también podemos usar el hecho de que `if` produce un resultado para escribir un código más corto:

```
signo <- if (x >= 0) "positivo" else "negativo"
signo
## [1] "positivo"
```

Supongamos ahora que tenemos una sentencia `if` múltiple, cuyas condiciones equivalen a ver si una expresión coincide con cierta cadena de caracteres. Por ejemplo:

```
operando1 <- 3
operando2 <- 4
operador <- readline("Operador (+, -, * o /: ")
resultado <- if (operador == '+') {
  operando1 + operando2
} else if (operador == '-') {
```

```
  operando1 - operando2
} else if (operador == '*') {
  operando1 * operando2
} else if (operador == '/') {
  operando1 / operando2
} else {
  "operando no contemplado"
}
resultado
```

En estas circunstancias: una selección múltiple en la que las distintas condiciones se corresponden con ver si una expresión coincide con una cadena de caracteres, se puede usar la instrucción `switch`:

```
operando1 <- 3
operando2 <- 4
operador <- readline("Operador (+, -, * o /: ")
resultado <- switch(operador,
                    '+' = operando1 + operando2,
                    '-' = operando1 - operando2,
                    '*' = operando1 * operando2,
                    '/' = operando1 / operando2,
                    "operando no contemplado"
)
resultado
```

Como puede observarse, el uso de `switch` produce un código mucho más compacto. Dado que para poder emplear la sentencia `switch` se requiere unas condiciones poco comunes no profundizaremos más en su uso. El lector interesado puede buscar más información sobre ella.

9.4 Ejercicios

1. Dadas los siguientes expresiones lógicas trata de calcular si producirán el valor `TRUE` o `FALSE` al ser evaluadas. Evalúalas después y comprueba si has acertado. Para evaluarlas ten en cuenta la precedencia de los operadores lógicos: `!` tiene la mayor precedencia, seguido de `&&` y de `||`.

   ```
   i <- 6
   j <- 12
   (2*i) <= j
   (2*i-1) < j
   (i > 0) && (i <= 10)
   ```

```
(i > 25) || (i < 50 && j < 50)
(i < 4) || (j > 5)
!(i > 6)
```

2. Escribe un programa que lea dos números y determine cuál de ellos es el mayor.

3. Realice un programa que lea un valor entero y determine si se trata de un número par o impar. Sugerencia: un número es par si el resto de dividirlo entre dos es cero.

4. Escribe un programa que lea del teclado un carácter e indique en la pantalla si el carácter es una vocal minúscula o no.

5. Escribe un programa que lea del teclado un carácter e indique en la pantalla si el carácter es una vocal minúscula, es una vocal mayúscula o no es una vocal.

6. Escribe un programa que solicite una edad e indique en la pantalla si la edad introducida está en el rango [18,25].

7. Implementa un programa que lea los coeficientes del siguiente sistema de ecuaciones e imprima los valores que son solución para x e y:

$$ax + by = c$$
$$dx + ey = f$$

   El sistema puede resolverse utilizando las siguientes fórmulas:

$$x = \frac{ce - bf}{ae - bd}$$
$$y = \frac{af - cd}{ae - bd}$$

   Hay que tener en cuenta aquellos casos que no tienen solución, esto ocurre cuando las rectas representadas por las ecuaciones son paralelas (la solución al sistema de ecuaciones es el punto donde se cortan las dos rectas).

8. Escribe un programa que calcule las soluciones de una ecuación de segundo grado de la forma $ax^2 + bx + c = 0$, teniendo en cuenta que:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

   Si las soluciones son complejas no las calcules e indica un mensaje en la pantalla. También debes tener en cuenta que `a` puede valer 0, en cuyo caso debes resolver un sistema con una incógnita. Si `a` y `b` valen 0 no hay solución.

9. Realice un programa que lea del teclado las longitudes de los tres lados de un triángulo y muestre en la pantalla qué tipo de triángulo es de acuerdo con la siguiente casuística (`a` denota la longitud del lado más largo y `b` y `c` denotan la longitud de los otros dos lados):

   - Si $a \geq b + c$, no se trata de un triángulo
   - Si $a^2 = b^2 + c^2$, es un triángulo rectángulo
   - Si $a^2 < b^2 + c^2$, es un triángulo acutángulo
   - Si $a^2 > b^2 + c^2$, es un triángulo obtusángulo

10. Haz una versión del ejercicio anterior en el que se lean los 3 vértices del triángulo en lugar de la longitud de los lados.

11. Dado el siguiente *data frame*:

```
notas <- data.frame(nombre = paste("Alumno", 1:10),
                    acertadas = sample(30,
                                       size = 10,
                                       replace = TRUE
                                    )
)
notas$falladas = 30 - notas$acertadas
```

    Añádele un campo con la nota categórica, de forma que los alumnos que hayan acertado 15 o más respuestas tengan de nota `"pasa"` y los que tengan menos de 15 tengan de nota `"no pasa"`. Usa la función `ifelse` para hacer el cálculo.

12. Realiza un programa que solicite dos valores enteros que indican un rango (se puede introducir primero el menor y luego el mayor o a la inversa) y a continuación lea un tercer valor entero e indique si dicho valor pertenece o no al rango.

13. Implementa un programa que dada una fecha (día, mes y año) determine si es válida o no. Suponemos que la fecha es válida si el día es válido según el mes, el mes está en el intervalo (1, 12) y el año es positivo. Consideramos que un año es bisiesto cuando es múltiplo de 4.

Chapter 10

Estructuras iterativas

Las estructuras iterativas, también llamadas repetitivas o cíclicas, ejecutan un conjunto de instrucciones de manera reiterada. En este tema estudiaremos las distintas instrucciones de R relacionadas con la iteración.

10.1 La sentencia `for`

La instrucción `for` itera por los elementos de un vector o lista, ejecutando un conjunto de instrucciones en cada iteración. Su sintaxis es:

```
for (item in vector) {
  conjunto_de_instrucciones
}
```

El conjunto de instrucciones se va a ejecutar tantas veces como elementos tenga el vector. La variable `item` toma en cada ejecución el valor de un elemento del vector. Por ejemplo:

```
for (x in c(2, 7, 9)) {
  print(x)
}
## [1] 2
## [1] 7
## [1] 9
```

Observa que el conjunto de instrucciones se ha ejecutado una vez por cada elemento del vector y que la variable `x` ha ido tomando en cada ejecución uno de los valores del vector, del primero al último.

La instrucción `for` nos permite procesar los elementos de un vector. El siguiente ciclo, por ejemplo, recorre los elementos de un vector para sumarlos, algo que podríamos hacer de forma más breve con la función `sum`:

```
v <- c(2, 9, -23, 5.5)
suma <- 0
for (x in v) {
  suma <- suma + x
}
suma
## [1] -6.5
sum(v)
## [1] -6.5
```

Si nos interesa iterar por los índices de un vector podemos usar la función `seq_along` para generar sus índices:

```
v <- 6:10
seq_along(v)
## [1] 1 2 3 4 5
for (ind in seq_along(v)) {
  cat(ind, '--', v[ind], '\n')
}
## 1 -- 6
## 2 -- 7
## 3 -- 8
## 4 -- 9
## 5 -- 10
```

10.2 La sentencia `while`

La instrucción `for` ejecuta un conjunto de instrucciones un número fijo de veces (tantas como la longitud del vector o lista usado). La instrucción `while` permite ejecutar un conjunto de instrucciones un número indeterminado de veces, pues el conjunto de instrucciones se ejecutan mientras que se verifica una condición lógica. La sintaxis del ciclo `while`es la siguiente:

```
while (condicion_logica) {
  conjunto_de_instrucciones
}
```

El funcionamiento es sencillo. Cuando el flujo de ejecución de un programa llega a un ciclo `while`, se evalúa su condición lógica. Si el resultado de la evaluación es

verdadero, se ejecuta el conjunto de instrucciones asociado y se vuelve a evaluar la condición lógica. Así hasta que la condición lógica es falsa, entonces se pasa a ejecutar la instrucción que siga en el programa a la instrucción `while`. Como ejemplo, vamos a ver un fragmento de código que calcula si un número entero positivo es primo:

```
num <- 19 # queremos ver si num es primo
encontrado_divisor <- FALSE
posible_divisor <- 2
while (!encontrado_divisor && posible_divisor < num) {
  if (num %% posible_divisor == 0) { # ¿num múltiplo de posible_divisor?
    encontrado_divisor <- TRUE
  } else {
    posible_divisor <- posible_divisor + 1
  }
}
if (encontrado_divisor) {
  cat(num, "no es un número primo")
} else {
  cat(num, "es un número primo")
}
## 19 es un número primo
```

La idea utilizada en el programa para comprobar si un número `num` es primo es probar con todos los números desde el 2 hasta `num - 1` para ver si alguno de esos números divide a `num`. Si existe algún divisor en el rango $[2, num - 1]$, `num` no es primo, en otro caso sí lo es. Inicialmente no conocemos ningún divisor, por lo que la variable `encontrado_divisor` se inicia a `FALSE`. La condición del ciclo hace que se itere mientras que no se ha encontrado un divisor (`!encontrado_divisor`) y no hayamos agotado todos los posibles divisores del número. Mientras que se verifican estas dos condiciones se ejecutan las instrucciones asociadas al ciclo. Estas comprueban si el actual divisor (`posible_divisor`) divide exactamente al número. Si es así, se actualiza la variable `encontrado_divisor` a `TRUE`, porque hemos hallado un divisor. En otro caso se incrementa en uno la variable `posible_divisor` para que en la siguiente iteración se pruebe con otro divisor. El ciclo termina cuando `encontrado_divisor` es verdadero (se ha encontrado un divisor, luego el número no es primo) o la variable `posible_divisor` es igual a `num` (no se ha encontrado ningún divisor, luego el número es primo).

La instrucción `while` es más general que la instrucción `for`. Cualquier cómputo que pueda realizarse con un `for` se puede hacer con un `while`, pero no a la inversa. Como ejemplo, vamos a sumar los elementos de un vector usando un ciclo `while`:

```
v <- c(2, 9, -23, 5.5)
suma <- 0
ind <- 1     # índice para recorrer los elementos de v
while (ind <= length(v)) {
  suma <- suma + v[ind]
  ind <- ind + 1
}
suma
## [1] -6.5
sum(v)
## [1] -6.5
```

No obstante, para recorrer los elementos de un vector es más elegante usar un `for`, pues la sentencia `for` está expresamente pensada para ello.

Si en el ciclo anterior se olvida escribir la última instrucción del conjunto de instrucciones: `ind <- ind + 1` tendríamos lo que se conoce como un *ciclo infinito*. Es decir, un ciclo que se ejecuta indefinidamente. Esto ocurre cuando en un ciclo el conjunto de instrucciones asociado no hace las modificaciones oportunas en variables para que la condición del ciclo se vuelva falsa. Puedes comentar la instrucción `ind <- ind + 1` y observa qué ocurre.

10.2.1 Lectura con centinela

La lectura con centinela es un tipo de lectura de datos en la que el usuario introduce un valor especial, llamado *centinela*, para indicar que no quiere introducir más datos. Veamos un ejemplo con un programa que acumula los números solicitados al usuario. El usuario debe introducir el valor cero cuando no quiera introducir más datos, es decir, el centinela es el cero.

```
# Lectura con centinela
suma <-  0
n <-  as.numeric(readline("Introduce número (0 para acabar): "))
while (n != 0) {
  suma <- suma + n
  n <- as.numeric(readline("Introduce número (0 para acabar): "))
}
cat('La suma de los números es', suma)
```

Observa que la primera instrucción `readline` sólo se ejecuta una vez y sirve para leer el primer número. El resto de lecturas se realizan con la última sentencia del ciclo `while`. En R se puede obtener una versión más sencilla del programa usando `scan` (el centinela de `scan` es escribir una línea en blanco):

```
cat("Introduce valores (línea en blanco para terminar): ")
v <- scan()
cat('La suma de los números es', sum(v))
```

10.2.2 Lectura con verificación

El ciclo `while` también se puede usar para comprobar de forma reiterada si la entrada proporcionada por el usuario se ajusta a unas especificaciones. Por ejemplo, supongamos que leemos una edad y queremos asegurarnos de que la edad leída está en el rango [18, 60]:

```
# Lectura con verificación
edad <- as.numeric(readline("Introduce edad: "))
while (edad < 18 || edad > 60) {
  edad <- as.numeric(readline("Reintroduce edad (rango [18, 60]: "))
}
cat('La edad es', edad)
```

Mientras que la edad no cumple los criterios de estar en el rango [18, 60] se solicita su introducción.

10.3 Instrucciones `break` y `next`

Las instrucciones `break` y `next` están ligadas a la ejecución de los ciclos. La instrucción `break` sirve para salirse de un ciclo. En caso de que se encuentre en un ciclo anidado (uno dentro de otro) se saldrá del ciclo más interno. Recomendamos que esta instrucción se use con mesura. El ejemplo de ver si un número es primo se puede codificar usando `break` de la siguiente manera:

```
num <- 19
encontrado_divisor <- FALSE
posible_divisor <- 2
while (posible_divisor < num) {
  if (num %% posible_divisor == 0) { # ¿num múltiplo de posible_divisor?
    encontrado_divisor <- TRUE
    break
  }
  posible_divisor <- posible_divisor + 1
}
if (encontrado_divisor) {
  cat(num, "no es un número primo")
} else {
```

```
  cat(num, "es un número primo")
}
## 19 es un número primo
```

En el interior del ciclo, si se encuentra un divisor se actualiza la variable `encontrado_divisor` a `TRUE` y se usa `break` para salir directamente de la ejecución del ciclo, porque ya hemos encontrado un divisor y sabemos que el número no es primo.

La instrucción `next` se utiliza con mucha menos frecuencia que `break`. Esta instrucción termina la ejecución del bloque de código asociado al ciclo e inicia una nueva ejecución (en el caso del ciclo `while` vuelve a evaluar la condición lógica asociada al ciclo). Por ejemplo:

```
# Suma los elementos positivos de un vector
v <- c(-2, 3, 5, -4)
suma <- 0
for (x in v) {
  if (x < 0) {
    next
  }
  suma <- suma + x
}
suma
## [1] 8
```

Este ciclo suma los elementos no negativos de un vector. Observa que en el interior del ciclo se usa `next` para dejar de ejecutar instrucciones y pasar a la siguiente iteración del ciclo en caso de que el número sea negativo (por lo tanto, si se ejecuta `next` no se ejecutará la instrucción `suma <- suma + x`). En este caso, el uso de `next` no resulta particulamente interesante, el código se podría haber escrito así:

```
# Suma los elementos positivos de un vector
v <- c(-2, 3, 5, -4)
suma <- 0
for (x in v) {
  if (x > 0) {
    suma = suma + x
  }
}
suma
## [1] 8
```

Sin embargo, hay circunstancias en las que el uso de la instrucción `next` puede producir un código más legible.

10.4 La sentencia `repeat`

El sentencia `repeat` ejecuta repetida e incondicionalmente un conjunto de instrucciones. Para que un ciclo implementado con `repeat` no se convierta en un ciclo infinito debe ejecutar en algún momento una instrucción `break` o, si el ciclo forma parte del código de una función, la instrucción `return` (la codificación de funciones se estudiará en el siguiente capítulo).

```
x <- 1
repeat {
  print(x)
  if (x == 5) break
  x <- x + 1
}
## [1] 1
## [1] 2
## [1] 3
## [1] 4
## [1] 5
```

En este curso recomendamos usar ciclos `while` o `for` antes que un ciclo `repeat`.

10.5 Escribir ciclos en una línea

Si el código asociado a un ciclo sólo contiene una instrucción, no es preciso usar llaves para delimitar sus instrucciones. Por ejemplo:

```
for (x in 1:5)
  cat(x^2, "\n")
## 1
## 4
## 9
## 16
## 25
```

Incluso se puede escribir todo en una línea:

```
for (x in 1:5) cat(x^2, "\n")
## 1
## 4
## 9
## 16
## 25
```

10.6 Código vectorizado versus ciclos

R tiene muchas funciones y operadores que actúan sobre todos los elementos de un vector, matriz, etc. Esto implica que muchos cálculos que en algunos lenguajes de programación han de realizarse usando ciclos se pueden expresar en R usando estas funciones y operadores, sin necesidad de escribir un ciclo. Por ejemplo, el calcular si un número es primo se podría resolver con el siguiente código:

```
num <- 19
no_primo <- num > 2 && any(num %% 2:(num-1) == 0)
if (no_primo) {
  cat(num, "no es un número primo")
} else {
  cat(num, "es un número primo")
}
## 19 es un número primo
```

A este tipo de código que usa funciones, operadores e indexación para realizar cálculos sobre vectores y demás estructuras de datos se le llama *código vectorizado*. El código vectorizado es muy compacto. Vamos a analizar en detalle las operaciones realizadas en el ejemplo previo. En primer lugar se generan los posibles divisores de `num`:

```
2:(num-1)
##  [1]  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18
```

Después se genera un vector con el resto de dividir `num` entre los posibles divisores:

```
num %% 2:(num-1)
##  [1] 1 1 3 4 1 5 3 1 9 8 7 6 5 4 3 2 1
```

A continuación se comprueba si las divisiones son exactas:

```
num %% 2:(num-1) == 0
##  [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [13] FALSE FALSE FALSE FALSE FALSE
```

Por último, se comprueba si al menos hay un divisor:

```
any(num %% 2:(num-1) == 0)
## [1] FALSE
```

¿Qué código es mejor, el que usa ciclos o el vectorizado? No existe una respuesta simple. En términos de claridad el código vectorizado es más conciso, pero a veces es difícil de entender. En términos de eficiencia dependerá de cada caso y habrá que cronometrar para ver cuál es más rápido (habría que probar con distintos tamaños de datos para ver cómo escalan los códigos). En general, si los códigos realizan las mismas operaciones el código vectorizado es más rápido, pues las funciones como `sum`, `mean`, etc suelen estar compiladas y se ejecutan más rápido que el código equivalente en R con ciclos, que es interpretado. No obstante, a veces el código vectorizado, pese a su brevedad, realiza más operaciones. En el ejemplo de ver si un número es primo, el código con ciclos deja de buscar divisores al encontrar uno, lo que no se puede expresar en el código vectorizado.

A continuación mostramos otro ejemplo de código con ciclos frente a código vectorizado. Se trata de sumar los elementos positivos de un vector, un código que ya hemos visto previamente:

```
# Suma los elementos positivos de un vector
v <- c(-2, 3, 5, -4)

# Versión con ciclos
suma <- 0
for (x in v) {
  if (x > 0) {
    suma <- suma + x
  }
}
suma
## [1] 8
sum(v[v>0]) # versión vectorizada
## [1] 8
```

10.7 Ejercicios

1. Observa los siguientes fragmentos de código y piensa qué resultado producirán. Ejecútalos y comprueba si has acertado.

```
# Ciclo simple
var <- 0
for (x in 1:10) {
  var <- var + 1
}
var
```

```r
# Ciclo anidado
var <- 0
for (x in 1:10) {
  for (y in 1:5) {
    var <- var + 1
  }
}
var
```

2. Realiza un programa que solicite del teclado un entero del 1 al 10 y muestre en la pantalla su tabla de multiplicar.

3. Realice un programa que calcule y muestre en la pantalla la suma de los cuadrados de los números enteros del 1 al 10 (la solución es $385 = 1^2 + 2^2 + ... + 10^2$). Escribe una versión con ciclos y otra vectorizada.

4. Realice un programa que lea del teclado 10 números e indique en la pantalla si el número cero estaba entre los números leídos.

5. Realiza un programa que lea del teclado números hasta que se introduzca un cero (es decir, es una lectura con centinela donde el centinela es el valor 0). En ese momento el programa debe terminar y mostrar en la pantalla la cantidad de valores mayores que cero leídos.

6. Existen muchos métodos numéricos capaces de proporcionar aproximaciones al número π. Uno de ellos es el siguiente:

$$\pi = \sqrt{\sum_{i=1}^{\infty} \frac{6}{i^2}}$$

   Crea un programa que lea el grado de aproximación (número de términos de la sumatoria) y devuelva un valor aproximado de π. Escribe una versión con ciclos y otra vectorizada.

7. Escribe un programa que lea un número entero no negativo n y dibuje un triángulo rectángulo con base y altura n, como el que se muestra a continuación para $n = 4$. Observa que debe aparecer un espacio entre cada asterisco situado en la misma línea:

```
*
* *
* * *
* * * *
```

8. El algoritmo de Euclides es un procedimiento para calcular el máximo común divisor de dos números naturales. Los pasos son:

- Se divide el número mayor (M) entre el menor (m).
- Si:
  - La división es exacta, entonces m es el máximo común divisor.
  - La división no es exacta, entonces $mcd(M, m) = mcd(m, M\%\%m)$.

Por ejemplo, vamos a calcular el máximo común divisor de 93164 y 5826:

- 93164 entre 5826 es 15 y sobran 5774, luego $mcd(93164, 5826) = mcd(5826, 5774)$.
- 5826 entre 5774 es 1 y sobran 52, luego $mcd(5826, 5774) = mcd(5774, 52)$.
- 52 entre 2 es 26 y sobran 0, luego $mcd(52, 2) = 2 = mcd(93164, 5826)$.

Escribe un programa que implemente este algoritmo.

9. Realice un programa que solicite al usuario un entero positivo e indique en la pantalla si el entero leído es una potencia de dos.

10. Un número perfecto es un número natural que es igual a la suma de sus divisores positivos, sin incluirse él mismo. Por ejemplo, 6 es un número perfecto porque sus divisores positivos son: 1, 2 y 3 y $6 = 1 + 2 + 3$. El siguiente número perfecto es el 28. Escribe un programa que lea un número natural e indique si es perfecto o no. Haz una versión con ciclos y otra vectorizada.

11. Realiza un programa que calcule los cuatro primeros números perfectos. Nota: el resultado es 6, 28, 496, 8128.

12. Realiza un programa que, dado un vector (por ejemplo, `sample(10)`), muestre sus elementos del último al primero en la pantalla. Usa un ciclo que genere los índices en orden inverso para obtener los elementos del vector.

13. Realiza un programa que, dado el siguiente vector:

```
set.seed(4)
v <- sample(10)
```

Calcule el primer elemento del vector que es mayor al elemento que le precede en el vector.

14. Realiza un programa que, dada la siguiente matriz:

```
set.seed(8)
m <- matrix(sample(20), nrow = 2)
```

Calcule el primer elemento de la segunda fila de la matriz que es mayor que el elemento de la primera fila ubicado en la misma columna.

Chapter 11

Funciones

Una función permite escribir un fragmento de código parametrizado. De esta forma, es posible escribir un bloque de código y ejecutarlo para distintos datos. Una función puede considerarse un subprograma que resuelve una subtarea. Un motivo para utilizar funciones es que permiten estructurar u organizar el código de un programa. Cuando hay que resolver un problema complejo, en lugar de intentar solucionarlo mediante un programa muy extenso es mejor descomponerlo en subproblemas. Los subproblemas deben tener una complejidad moderada, de forma que sean resueltos por subprogramas (como las funciones) sencillos. Así, en lugar de utilizar un programa muy grande para resolver un problema complejo se emplean distintos subprogramas que resuelven tareas sencillas y que se combinan para producir una solución final más simple.

En temas previos ya hemos usado muchas funciones de R, en este tema vamos a aprender a crear nuestras propias funciones.

11.1 Creación de funciones

Vamos a empezar a estudiar las funciones creando una función que calcula el máximo de sus dos argumentos:

```
maximo <- function(a, b) {
  if (a > b) {
    m <- a
  } else {
    m <- b
  }
  m
}
```

```
maximo(6, 2)
## [1] 6
y <- 18
maximo(20, y + 7)
## [1] 25
```

Para crear una función se usa la palabra reservada `function`. A continuación se enumeran los parámetros formales de la función separados por comas y encerrados entre paréntesis (en el ejemplo los parámetros formales son `a` y `b`). Después viene el código de la función. Puesto que el código del ejemplo tiene más de una instrucción, hay que encerrarlo entre llaves. Con esto se crea un objeto función que se asigna a la variable `maximo`.

Una vez creado el objeto función y haber sido asignado a `maximo`, en el ejemplo aparecen dos invocaciones a la función. En la primera se usan los parámetros reales 6 y 2, y en la segunda 20 y la expresión `y + 7`. Al invocar a una función se evalúan las expresiones que conforman los parámetros reales y se emparejan con los parámetros formales, es decir, a cada parámetro formal se le asigna el resultado de evaluar la expresión usada como parámetro real. Los parámetros formales son variables que reciben la información de entrada de la función (en el ejemplo, `a` y `b`). Una vez emparejados parámetros reales con formales se ejecutan las instrucciones de la función. Una función devuelve como salida el resultado de evaluar la última instrucción que ejecuta. En el ejemplo, la última instrucción ejecutada es `m`, que es una expresión que produce el máximo de los dos valores de entrada de la función (almacenados en `a` y `b`). Veamos otros ejemplos de invocación a la función `maximo`:

```
m <- maximo(2, 3)  # se asigna la salida de la función a m
2 + maximo(4, 5.2) # la invocación a la función forma parte de una expresión
## [1] 7.2
```

11.2 La función `return`

En la sección anterior hemos comentado que una función devuelve el resultado de evaluar la última instrucción que ejecuta. Esto es cierto si el código de la función no invoca a la función `return`. La función `return` tiene un parámetro y produce el siguiente efecto: termina la ejecución de la función en la que se ejecuta y el valor devuelto por la función es el parámetro con que es invocada `return`. Por ejemplo, la función `maximo` previa puede codificarse así:

```
maximo <- function(a, b) {
  if (a > b) {
    m <- a
```

```
  } else {
    m <- b
  }
  return(m)
}
maximo(6, 2)
## [1] 6
```

Otra posibilidad es:

```
maximo <- function(a, b) {
  if (a > b) {
    return(a)
  } else {
    return(b)
  }
}
maximo(6, 7)
## [1] 7
```

O incluso:

```
maximo <- function(a, b) {
  if (a > b) {
    return(a)
  }
  return(b)
}
maximo(6, 2)
## [1] 6
```

En este último ejemplo, la instrucción `return(b)` se ejecuta sólamente si `a` no es mayor que `b`. Si `a` fuera mayor que `b` entonces se ejecutaría `return(a)`, lo que terminaría la ejecución de la función e impediría que se ejecutara cualquier instrucción tras el `if`.

11.3 Variables locales

Una función puede realizar cálculos complejos, por lo que a veces necesita usar variables para almacenar los resultados intermedios de sus cálculos. Por ejemplo, la siguiente función calcula la suma de los elementos de un vector:

```
# Función que suma los elementos de un vector (recibido en v)
suma <- function(v) {
  s <- 0
  for (x in v) {
    s <- s + x
  }
  s
}
suma(c(2, 5, -3))
## [1] 4
```

Las variables creadas en una función se llaman *variables locales*. En el ejemplo, la variable `s` es una variable local que se usa para acumular los elementos del vector. Las variables locales, junto con los parámetros formales, se crean al ejecutarse la función y normalmente desaparecen al terminar su ejecución. Las variables locales y parámetros formales son independientes de otras variables que puedan existir fuera de la función con el mismo nombre. Por ejemplo:

```
v <- 5
s <- 10
suma <- function(v) {
  s <- 0
  for (x in v) {
    s <- s + x
  }
  s
}
suma(c(2, 5, -3))
## [1] 4
v
## [1] 5
s
## [1] 10
```

Observa que el parámetro formal `v` y la variable local `s` coinciden en nombre con otras variables externas a la función. Al invocar a la función `suma` se crean las variables locales `v` y `s` y al terminar su ejecución desaparecen, siendo independientes de las variables `v` y `s` que conservan sus valores.

11.4 Variables globales

Una función también puede usar variables que se han creado fuera de la función. Por ejemplo:

```
w <- 10
f <- function(x) {
  x + w
}
f(4)
## [1] 14
```

La función `f` usa el valor de la variable `w` que no es una variable local, sino que es global a la función (ha sido creada fuera). En general, el uso de variables globales no es considerada una buena práctica de programación, por lo que no las usaremos. No obstante, es conveniente saber de su existencia, pues hay muchos programas en R que las utilizan.

El operador de superasignación: `<<-`, permite modificar variables globales:

```
w <- 10
f <- function() {
  w <<- w + 2
}
w
## [1] 10
f()
w
## [1] 12
```

Puesto que nosotros no usaremos variables globales, no usaremos este operador.

11.5 Funciones que devuelven varios valores

Las funciones de ejemplo que hemos visto hasta ahora devuelven un escalar. No obstante, algunas funciones deben devolver más de una valor. Sin embargo, las funciones de R sólo pueden devolver un objeto, resultado de evaluar una expresión. Por lo tanto, cuando queramos devolver más de un dato habrá que devolver un objeto que almacene una colección de datos, como un vector, factor, matriz, lista o *data frame*. Habrá que elegir el tipo de objeto que creamos que sea más adecuado a nuestros cálculos. Por ejemplo, la función `range` devuelve el mínimo y el máximo de los elementos de un vector. Vamos a implementar nuestra propia versión:

```
# rango de los valores de un vector
# Parámetro de entrada: v, el vector
rango <- function(v) {
  c(min(v), max(v))
```

```
}
(v <- sample(10, 4))
## [1]  7  8  3 10
rango(v)
## [1]  3 10
range(v) # para comparar si funciona igual que range
## [1]  3 10
```

Hemos decidido devolver un vector con dos elementos: el mínimo y el máximo. Sin embargo, también podíamos haber optado por devolver una lista con el mínimo y el máximo:

```
# rango de los valores de un vector
rango2 <- function(v) {
  list(minimo = min(v), maximo = max(v))
}
rango2(v)
## $minimo
## [1] 3
##
## $maximo
## [1] 10
range(v)
## [1]  3 10
```

En general, las funciones que devuelven datos de distinto tipo usan una lista como valor de retorno, pues las listas permiten incluir datos de distintos tipos.

11.6 Parámetros por defecto e invocación por nombre

En el tema de variables y expresiones ya comentamos que las funciones pueden tener parámetros por defecto. Un parámetro por defecto es un parámetro formal que se define terminándolo con un igual y una expresión. Vamos a ver su utilidad con un ejemplo. La siguiente función sirve para calcular la distancia euclídea entre dos puntos en un espacio bidimensional y no usa parámetros por defecto:

```
# distancia entre los puntos (x1, y1) y (x2, y2)
distancia_eu <- function(x1, y1, x2, y2) {
  sqrt((x1 - x2)^2 + (y1 - y2)^2)
}
distancia_eu(1, 1, 2, 2) # distancia entre (1, 1) y (2, 2)
## [1] 1.414214
```

Como es muy común calcular la distancia de un punto al origen, se decide que el segundo punto sea, por defecto, el origen:

```
# distancia entre los puntos (x1, y1) y (x2, y2)
distancia_eu2 <- function(x1, y1, x2 = 0, y2 = 0) {
  sqrt((x1 - x2)^2 + (y1 - y2)^2)
}
distancia_eu2(1, 1, 2, 2) # distancia entre (1, 1) y (2, 2)
## [1] 1.414214
distancia_eu2(4, 4)       # distancia entre (4, 4) y (0, 0)
## [1] 5.656854
```

En esta versión de la función con parámetros por defecto, cuando un punto sea el origen no es preciso especificarlo. El efecto de un parámetro por defecto es que, si no se especifica al ser invocada la función, el parámetro formal toma el valor por defecto, es decir, el especificado tras el operador igual en la definición de la función.

En todas las invocaciones a funciones realizadas hasta ahora en este tema se ha usado la invocación posicional, es decir, parámetros reales y formales se emparejan según su posición (el primer parámetro real se empareja con el primer parámetro formal, el segundo con el segundo, …). Recordemos que al invocar una función en R los parámetros también se pueden especificar por nombre:

```
# Invocación posicional: distancia entre (1, 1) y (2, 2)
distancia_eu2(1, 1, 2, 2)
## [1] 1.414214
# Invocación por nombre: distancia entre (1, 1) y (2, 2)
distancia_eu2(x1 = 1, x2 = 2, y1 = 1, y2 = 2)
## [1] 1.414214
```

11.7 Las funciones son objetos

En R una función es un objeto más, como un vector o un *data frame*. Esto implica que se pueda operar con funciones como con otros tipos de objetos. En el siguiente ejemplo, se asigna una función:

```
f <- function(x) {
  x^2
}
f(8)
## [1] 64
g <- f # se asigna a g la función asociada a f
g(3)   # invocamos a la función a través de g
## [1] 9
```

Otro ejemplo es crear una lista cuyos elementos son funciones:

```
suma   <- function(a, b) a + b
resta  <- function(a, b) a - b
operaciones <- list(s = suma, r = resta)
operaciones[[1]](2, 4)
## [1] 6
operaciones[[2]](2, 4)
## [1] -2
operaciones$s(5, 6)
## [1] 11
```

Una función también puede ser el parámetro de otra función. Ya hemos visto ejemplos de esto en funciones como `apply` o `lapply`. Vamos a crear nuestra versión simplificada de `lapply`. Recordemos que esta función aplica una función, recibida como parámetro, a los elementos de una lista y devuelve una lista con el resultado.

```
# Aplica una función a los elementos de una lista
# Parámetros de entrada:
#  - l, la lista
#  - f, la función a aplicar a los elementos de l
# Parámetros de salida: una lista con el resultado de aplicar f
#                       a los elementos de l
mi_lapply <- function(l, f) {
  salida <- list()
  for (ind in seq_along(l)) {
    salida[[ind]] <- f(l[[ind]])
  }
  return(salida)
}
l <- list(1, 2, 3)
f <- function(x) x^3
mi_lapply(l, f)
## [[1]]
## [1] 1
##
## [[2]]
## [1] 8
##
## [[3]]
## [1] 27
lapply(l, f)
## [[1]]
## [1] 1
```

```
##
## [[2]]
## [1] 8
##
## [[3]]
## [1] 27
```

Aunque todo esto es bastante útil, especialmente los parámetros de tipo función, en este curso no lo vamos a utilizar.

11.8 Funciones anónimas

Hasta ahora al definir las funciones siempre hemos asignado el objeto función creado a un identificador. Una *función anónima* es aquella que no tiene un identificador asociado. En R hay ocasiones en que sólo necesitamos una función temporalmente y podemos evitarnos el darle un nombre si usamos una función anónima. Por ejemplo, vamos a usar `sapply` para ver cuántos elementos distintos tiene cada variable (columna) del *data frame* `mtcars`:

```
f <- function(variable) {
  length(unique(variable))
}
sapply(mtcars, f)
##  mpg  cyl disp   hp drat   wt qsec   vs   am gear carb
##   25    3   27   22   22   29   30    2    2    3    6
```

Si observas la salida verás que sólo hay tres valores de cilindros distintos (variable `cyl`). La función `f` calcula el número de valores distintos de un vector. Para ello usa la función `unique` que elimina repetidos y depués cuenta cuántos elementos (sin repetidos) hay. Si la función `f` sólo es usada para este cálculo con `sapply`, podemos evitarnos el darle un nombre:

```
# usando una función anónima
sapply(mtcars, function(x) length(unique(x)))
##  mpg  cyl disp   hp drat   wt qsec   vs   am gear carb
##   25    3   27   22   22   29   30    2    2    3    6
```

Las funciones anónimas suelen ser funciones cortas que se pueden escribir en una única línea. Como segundo ejemplo de función anónima vamos a usar la función `integrate`, que integra a la función que le pasemos como primer parámetro en el intervalo especificado con los siguientes dos parámetros. Por ejemplo, una aproximación a la integral de $2x^2 + 7$ en el intervalo $[0, 5]$ es:

```
f <- function(x) {
  2*x^2 + 7
}
integrate(f, 0, 5)
## 118.3333 with absolute error < 1.3e-12
```

De nuevo, podemos evitar dar nombre a la función si no la precisamos para otros cálculos:

```
integrate(function(x) 2*x^2 + 7, 0, 5)
## 118.3333 with absolute error < 1.3e-12
```

Otro ejemplo en el que se podría usar las funciones anónimas es en el código previo en el que se creaba una lista de funciones, éstas pueden ser anónimas:

```
operaciones <- list(suma  = function(a, b) a + b,
                    resta = function(a, b) a - b
)
operaciones[[1]](2, 4)
## [1] 6
operaciones[[2]](2, 4)
## [1] -2
operaciones$suma(5, 6)
## [1] 11
```

11.9 Funciones con un número variable de parámetros

Hay funciones, como `sum`, que toman un número variable de parámetros. En concreto, `sum` suma todos los valores que le pasemos:

```
sum(1, 2)
## [1] 3
sum(1, 2, 3)
## [1] 6
sum(1, c(3, 8))
## [1] 12
```

Para implementar funciones con un número variable de parámetros hay que usar un parámetro especial que viene dado por tres puntos. Dentro del código de la función podemos utilizar `list(...)` para obtener una lista con los parámetros con que se ha invocado la función. Vamos a implementar una versión simplificada de `sum` que sólo suma escalares:

```r
# Función que suma un número variable de escalares
suma <- function(...) {
  l <- list(...)
  s <- 0
  for (x in l) {
    s <- s + x
  }
  return(s)
}
suma(1, 2)
## [1] 3
suma(1, 2, 3)
## [1] 6
```

11.10 Ejercicios

1. Realiza una función que reciba como parámetro un número y devuelva su valor absoluto. Nota: en R existe una versión más general de esta función llamada `abs`.

2. Escribe una función que reciba un vector de números y devuelva una lista con los elementos positivos y los elementos negativos del vector.

3. Realiza una versión de la función `sign`. Esta función recibe un vector numérico y devuelve un vector del mismo tamaño que el vector original, pero con los valores 1 (valores positivos), 0 (valores 0) y -1 (valores negativos). Por ejemplo, `sign(c(-4, 2, 0, 3))` devuelve el vector `c(-1, 1, 0, 1)`.

4. La función `choose` permite calcular un número combinatorio. Por ejemplo, `choose(5, 2)` calcula $\binom{5}{2}$. Escribe una versión de `choose`. En la implementación puedes usar la función `factorial`.

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

5. R contiene una serie de funciones que implementan las operaciones entre conjuntos: unión, intersección, diferencia e igualdad:

```r
x <- c(1, 2, 5)
y <- c(5, 6, 1, 3)
union(x, y)      # unión
## [1] 1 2 5 6 3
intersect(x, y)  # intersección
```

```
## [1] 1 5
setdiff(x, y)    # Diferencia: elementos de x que no están en y
## [1] 2
setdiff(y, x)
## [1] 6 3
setequal(x, y)   # igualdad
## [1] FALSE
setequal(x, c(5, 2, 1))
## [1] TRUE
```

Implementa versiones de estas funciones. En la implementación puedes usar el operador %in% que comprueba si un elemento está en un vector:

```
2 %in% x
## [1] TRUE
c(2, 4) %in% x
## [1]  TRUE FALSE
```

6. Siguiendo con el ejercicio previo, implementa una función que calcule si un conjunto es un subconjunto de otro y una función que calcule la diferencia simétrica, es decir, aquellos elementos que pertenecen a uno de los conjuntos, pero no a ambos a la vez.

7. Escribe una función que devuelva el valor asociado a una función matemática definida por intervalos (mira la solución para ver una representación gráfica de la función):

$$f(x) = \begin{cases} x^2 & \text{si } x < 0 \\ seno(x) & \text{si } x \in [0, 2\pi] \\ x - 2\pi & \text{si } x > 2\pi \end{cases}$$

8. Escribe una función que dada una matriz permute sus filas. Es decir, la primera fila se debe intercambiar con la última, la segunda fila con al penúltima y así sucesivamente. Como ejemplo, dada la matriz:

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

se debe devolver la matriz:

$$\begin{pmatrix} 7 & 8 & 9 \\ 4 & 5 & 6 \\ 1 & 2 & 3 \end{pmatrix}$$

9. Escribe una función que dado un valor y un vector devuelva cuántas veces se encuentra el valor en el vector.

10. Realiza una función que dado un vector de cadenas de caracteres devuelva la cadena más larga y la más corta (si hay varias igual de largas o cortas se puede elegir cualquiera). Para resolver este ejercicio pueden resultar útiles las funciones `which.max` y `which.min`. La función `nchar` devuelve el número de caracteres de una cadena de caracteres.

11. Una matriz es simétrica si es cuadrada y es igual a su traspuesta. Escribe una función que devuelva un valor lógico indicando si una matriz es simétrica.

12. Realiza una función que dado un vector numérico indique si existe algún elemento del vector que es igual a la suma del resto de elementos del vector.

13. Una función recibe como parámetro una matriz. La matriz representa los tiempos empleados por un ciclista en varias etapas. Cada fila representa una etapa. La primera columna de la matriz almacena el número de horas, la segunda columna el número de minutos y la tercera columna el número de segundos que tardó en completar la etapa. Por ejemplo, si se recibe:

$$\begin{pmatrix} 2 & 30 & 50 \\ 1 & 55 & 20 \end{pmatrix}$$

El ciclista ha completado dos etapas. En la primera etapa ha tardado 2 horas, 30 minutos y 50 segundos. La función debe devolver una lista con el número total de horas, minutos y segundos empleados por el ciclista en cubrir el total de etapas. Para los datos de ejemplo se devolvería 4 horas, 26 minutos y 10 segundos.

14. Realiza una función que reciba como parámetro una matriz y devuelva las posiciones de los elementos que sean a la vez mínimos de su columna y máximos de su fila. Por ejemplo, dada la matriz:

$$\begin{pmatrix} 7 & 8 & 9 \\ 4 & 5 & 6 \\ 1 & 2 & 3 \end{pmatrix}$$

devolvería la posición (3, 3).

15. Realiza un programa que permita gestionar las reservas de una sala de cine para una sesión. La sala tiene 9 filas con 9 asientos por fila. Los asientos se numeran del 11 al 19 para la primera fila, del 21 al 29 para la segunda fila y así sucesivamente. La sala puede representarse como una matriz. Crea las siguientes funciones:

- Función que inicie la matriz. Inicialmente todos los asientos deben estar libres.
- Función que muestre en la pantalla la ocupación de la sala.
- Función que devuelva cuántos asientos libres hay en la sala.
- Función que dado un número de entradas devuelva un vector con los números de las filas en que existe, como mínimo, ese número de asientos libres.
- Función que dado un número de fila y un entero positivo n reserve n asientos en la fila especificada. Devuelve un vector con los números de los asientos reservados. Puedes suponer que el número de fila es correcto y que la fila contiene esa cantidad de asientos libres.
- Función que dado un entero positivo n reserve n asientos en la sala. Si no existen n asientos en la sala no se debe reservar ninguno. Devuelve un vector con los números de los asientos reservados.

Escribe un programa que solicite cantidades de entradas. Si la cantidad de entradas es mayor a la cantidad de asientos libres indíquelo en la pantalla. Si es menor o igual reserve esas entradas. Proceda solicitando cantidades de entradas y reservando hasta que no haya entradas libres en la sala o se introduzca una cantidad de entradas negativa. Reserve entradas de la siguiente forma:

- Si es posible albergar todas las entradas en la misma fila, reserve la fila de mayor numeración con capacidad para albergar el número de entradas solicitadas.
- En otro caso obtenga los asientos de cualquier lugar disponible.

Cada vez que se reserven entradas muestre los números de entradas reservadas y la ocupación global de la sala. Por ejemplo:

```
1: L L L L L L L L L
2: L L L L L L L L L
3: L L L L L L L L L
4: L L L L L L L L L
5: L L L L L L L L L
6: L L L L L L L L L
7: L L L L L L L L L
8: L L L L L L L L L
9: L L L L L L L L L
```

Cantidad de entradas: 8 Asientos reservados: [91, 92, 93, 94, 95, 96, 97, 98]

```
1: L L L L L L L L L
2: L L L L L L L L L
3: L L L L L L L L L
```

```
4: L L L L L L L L L L
5: L L L L L L L L L L
6: L L L L L L L L L L
7: L L L L L L L L L L
8: L L L L L L L L L L
9: O O O O O O O O O L
```

Cantidad de entradas: 11 Asientos reservados: [99, 81, 82, 83, 84, 85, 86, 87, 88, 89, 71]

```
1: L L L L L L L L L L
2: L L L L L L L L L L
3: L L L L L L L L L L
4: L L L L L L L L L L
5: L L L L L L L L L L
6: L L L L L L L L L L
7: O L L L L L L L L L
8: O O O O O O O O O O
9: O O O O O O O O O O
```

Cantidad de entradas:

Chapter 12

Graficos

En este tema se bosquejan las posibilidades gráficas de R. En concreto vamos a describir algunas de las funciones que incluye el núcleo de R para trabajar con gráficos. Las funciones gráficas de R suelen ser genéricas y tienen un gran número de opciones. Por lo tanto, aquí nos limitaremos a comentar sus usos más comunes. Si estás interesado en conocer de manera exhaustiva el funcionamiento de alguna función consulta su ayuda.

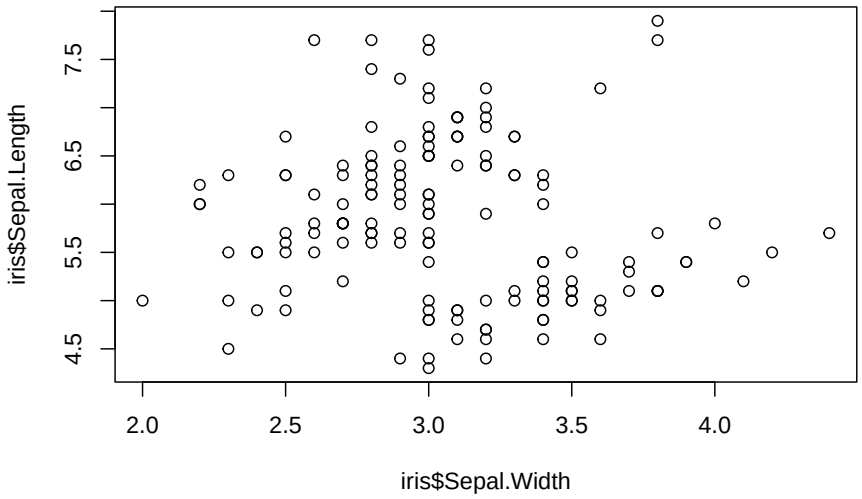
12.1 La función `plot`

La función `plot` es una de las funciones más usadas para generar gráficos en R. Se trata de una función genérica, por lo que puede ser invocada para representar gráficamente distintos tipos de datos. La función `plot` es de nivel alto, es decir, genera un nuevo gráfico al ser invocada (las funciones de nivel bajo añaden información a un gráfico ya existente). En las siguientes secciones vamos a analizar algunas posibilidades de esta función.

12.1.1 Diagrama de dispersión

La función `plot` permite generar un diagrama de dispersión (*scatter plot*). Veamos un ejemplo:

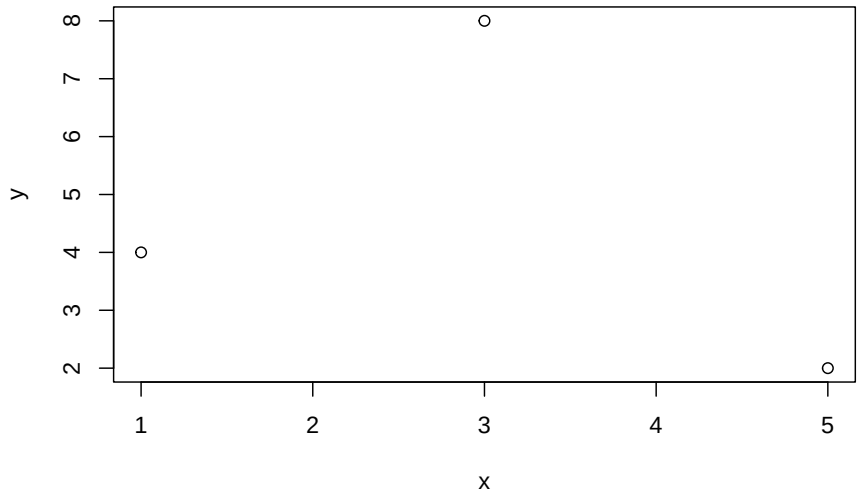
```
plot(iris$Sepal.Width, iris$Sepal.Length)
```



En este caso se muestra la longitud de sépalo frente a la anchura de sépalo de las flores del *data frame* `iris`. Se han usado dos vectores para especificar los datos, pero también se permite especificar los datos mediante una lista con componentes `x` e `y` o mediante una matriz con dos columnas.

Veamos otro ejemplo en el que se crean los puntos (datos) a visualizar:

```
x <- c(1, 5, 3)
y <- c(4, 2, 8)
plot(x, y)
```

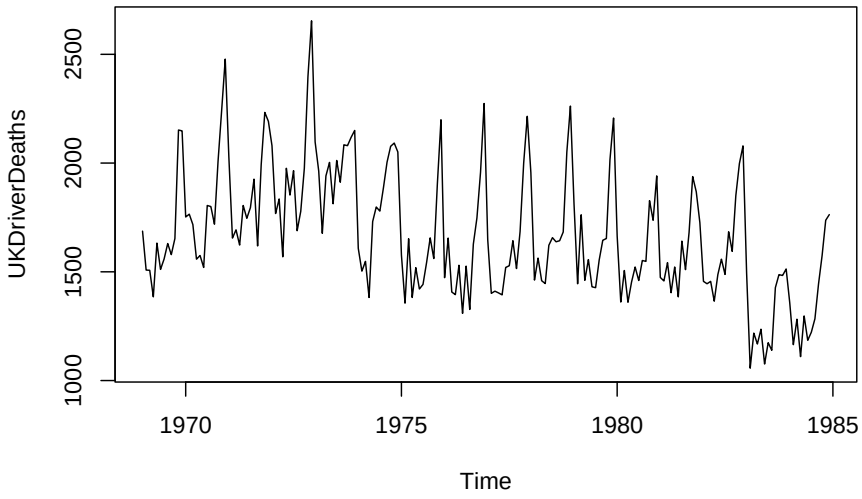


En este último ejemplo se visualizan los puntos (1,4), (5, 2) y (3, 8).

12.1.2 Serie temporal

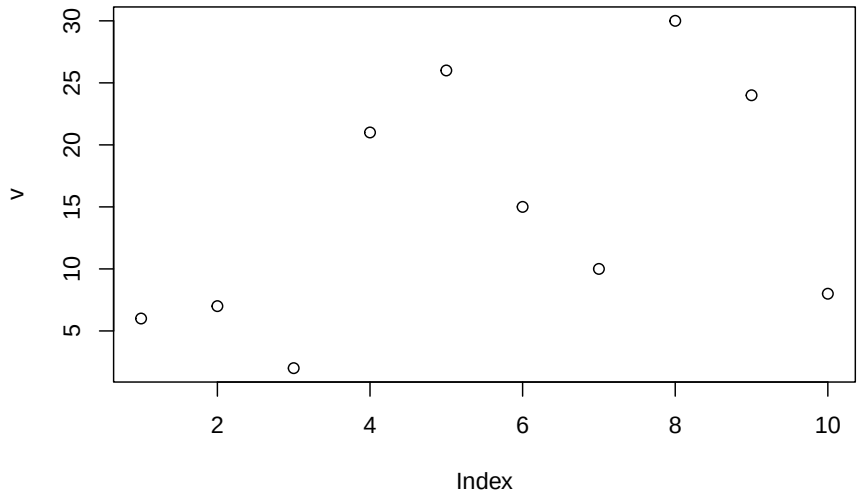
Una serie temporal es una secuencia de datos medidos en determinados momentos, generalmente equidistantes en el tiempo, y ordenados cronológicamente. `plot` nos permite visualizar una serie temporal:

```
plot(UKDriverDeaths)
```



Si usamos un vector como fuente de datos, se muestra los valores del vector frente a sus índices:

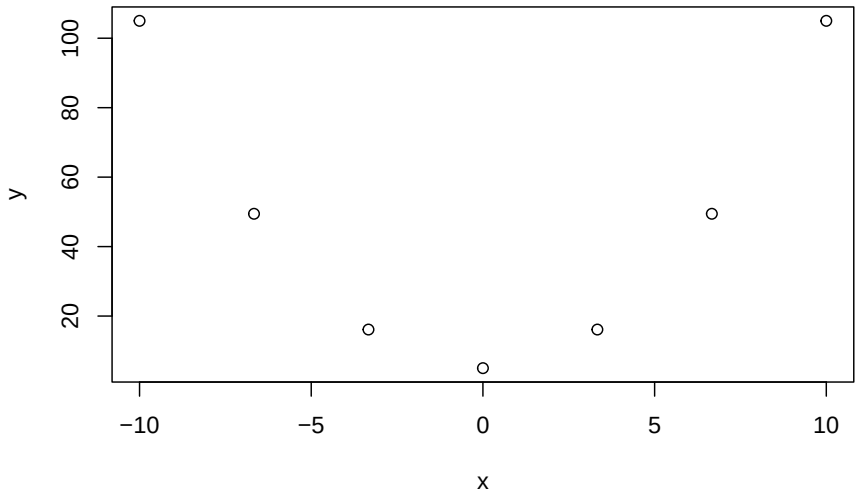
```
(v <- sample(30, size = 10))
##  [1]  6  7  2 21 26 15 10 30 24  8
plot(v)
```



12.1.3 Visualizar una función

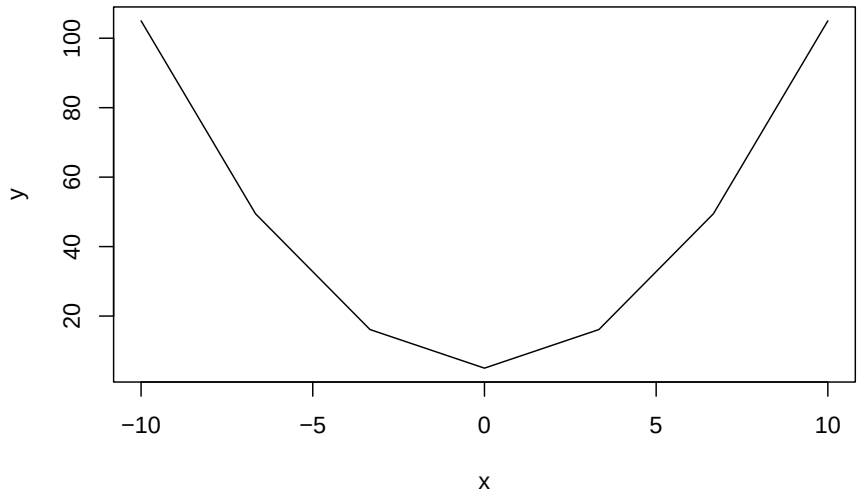
Podemos visualizar los valores de una función real de variable real si almacenamos en un vector una serie de valores de `x` (en orden creciente) y en otro vector sus correspondientes valores de `y` para la función. Por ejemplo, vamos a visualizar la función $y = x^2 + 5$ para 7 valores equidistantes de `x` en el intervalo [-10, 10]:

```
x <- seq(-10, 10, length.out = 7)
y <- x^2 + 5
plot(x, y)
```

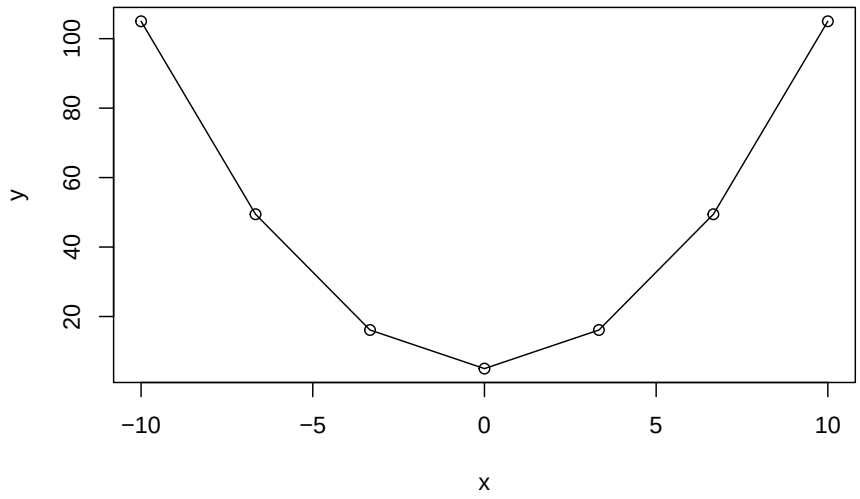


El parámetro `type` de `plot` permite visualizar los datos de distintas formas. Por defecto, este parámetro vale `"p"` y dibuja los puntos especificados, pero también se puede dibujar las líneas que unen los puntos, con `"l"`, o líneas y puntos con `"b"` (hay más posibilidades):

```
plot(x, y, type = "l") # visualizamos las líneas que unen los puntos
```

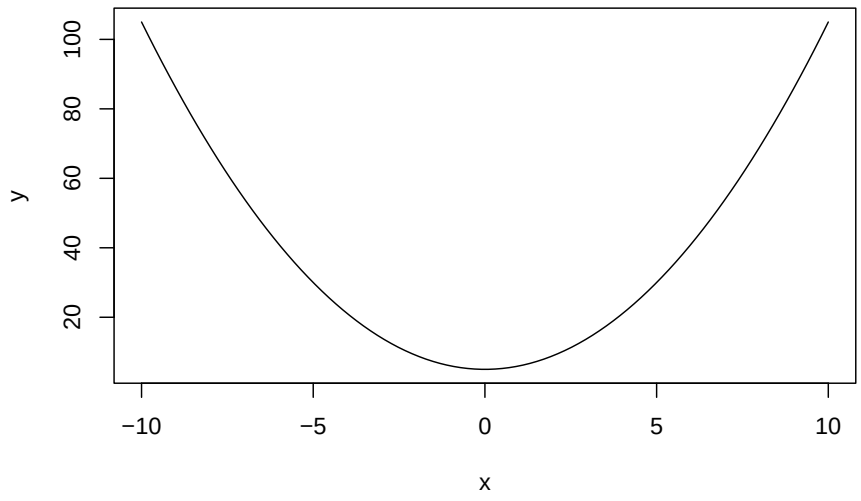



```
plot(x, y, "o") # líneas y puntos superpuestos
```



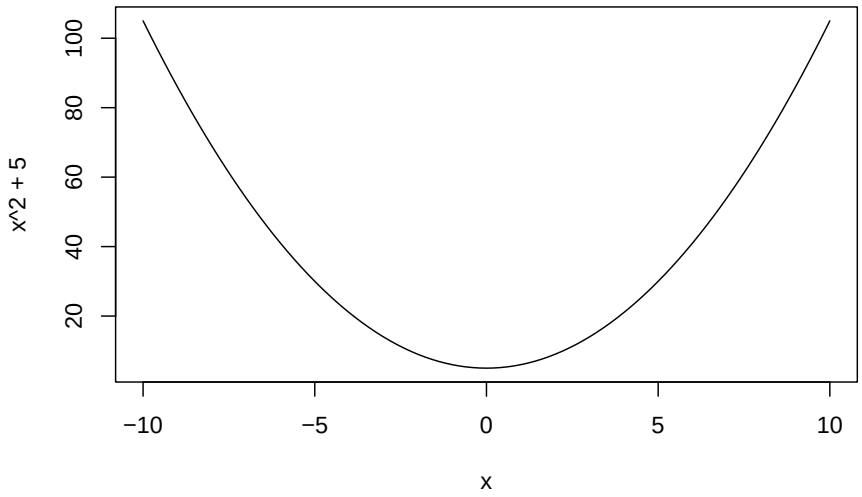
Para visualizar una función con más precisión basta con especificar más puntos de la función:

```
x <- seq(-10, 10, length.out = 100)
y <- x^2 + 5
plot(x, y, type = "l")
```



La función `curve` es una alternativa a `plot` para visualizar la curva asociada a una función. Consulta su ayuda para ver todas las posibilidades:

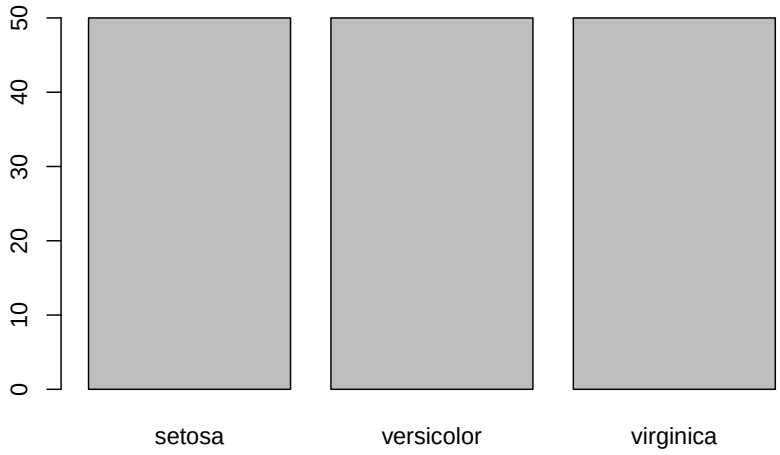
```
curve(x^2 + 5, from = -10, to = 10)
```



12.1.4 Factores y diagramas de caja

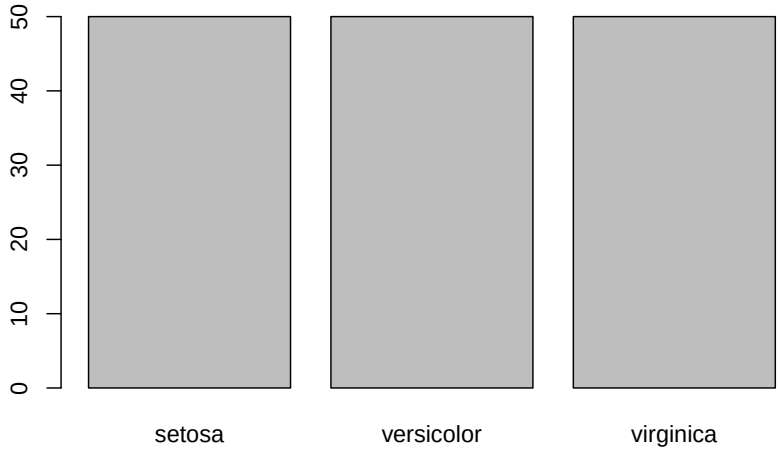
En el caso de que le pasemos a `plot` un factor obtendremos un diagrama de barras con las frecuencias de las distintas categorías:

```
plot(iris$Species)
```



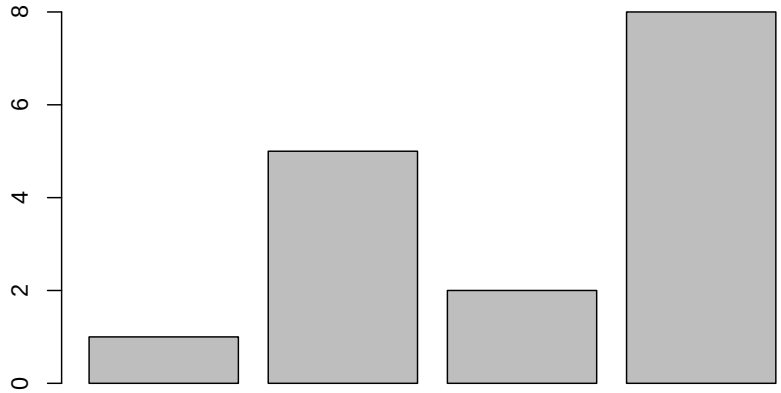
Podemos observar que existen 50 datos de las tres especies de lirio del conjunto de datos `iris`. Podríamos obtener un resultado similar con la función `barplot`:

```
barplot(table(iris$Species))
```



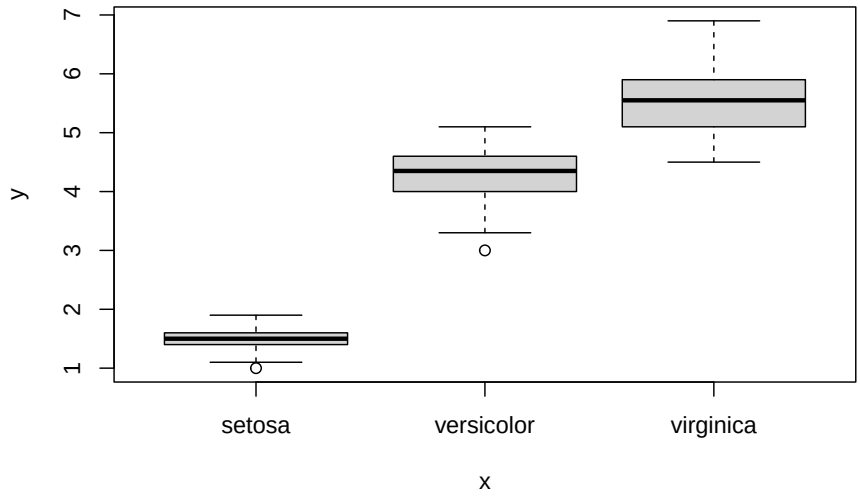
La función `barplot` también funciona con vectores:

```
barplot(c(1, 5, 2, 8))
```



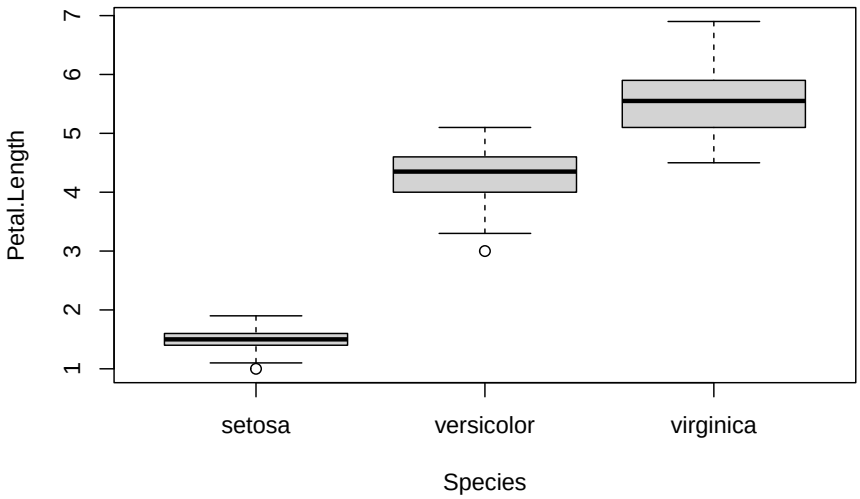
Si usamos un factor y un vector numérico, se obtiene un diagrama de cajas (o de cajas y bigotes o *box plot*) por cada categoría del factor. Por ejemplo, veamos la distribución de la longitud de pétalo para las distintas categorías de lirios:

```
plot(iris$Species, iris$Petal.Length)
```



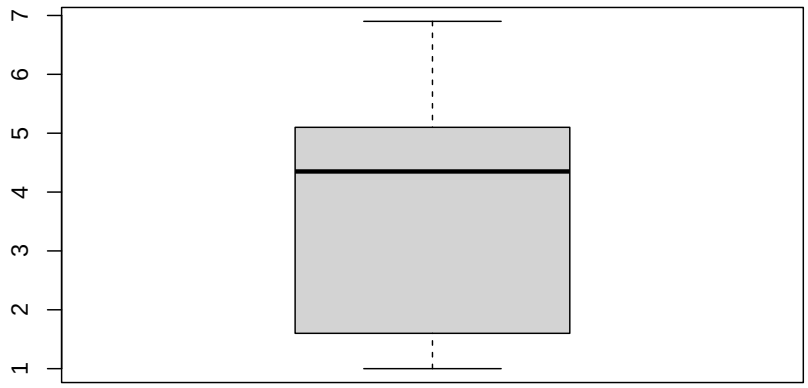
La función `boxplot` es una alternativa a `plot` para visualizar diagramas de cajas:

```
boxplot(Petal.Length ~ Species, data = iris)
```



`boxplot` también permite generar un diagrama de cajas con los elementos de un vector. Por ejemplo, veamos el diagrama de cajas de la longitud de pétalo de todas las flores de `iris`:

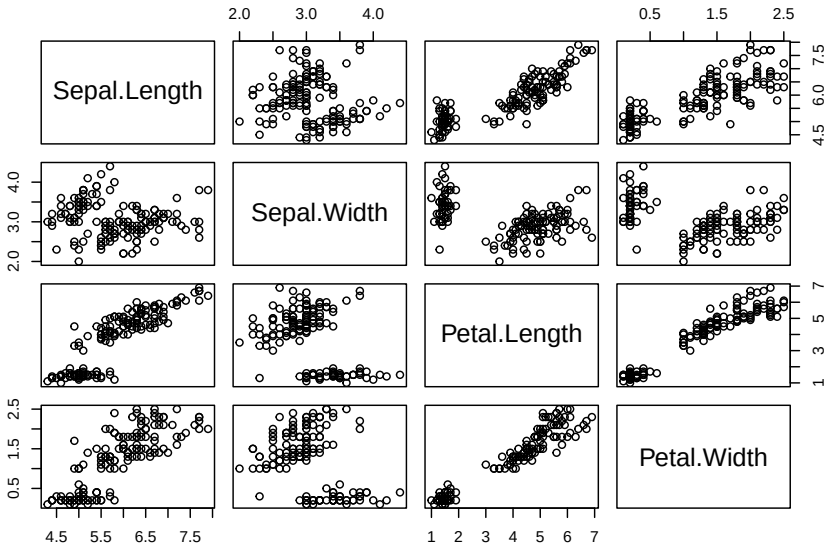
```
boxplot(iris$Petal.Length)
```



12.1.5 Diagramas de dispersión emparejados

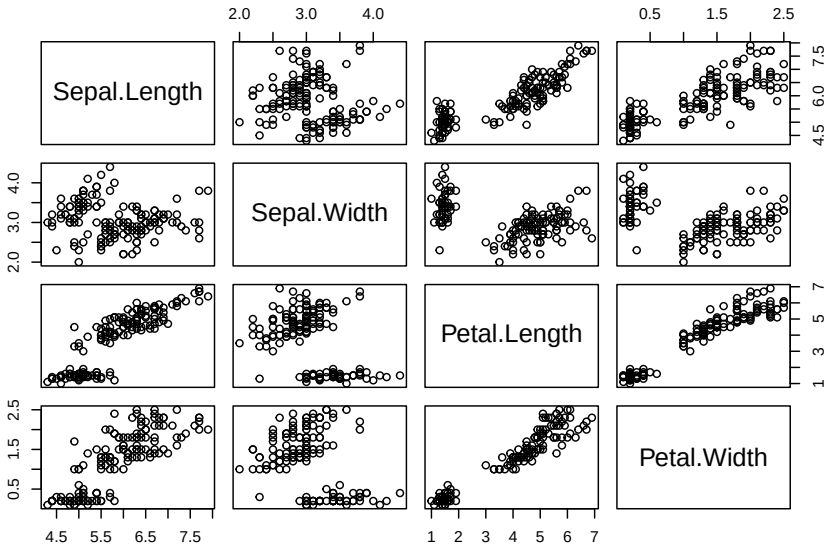
Cuando `plot` recibe como parámetro un *data frame* genera diagramas de dispersión para todos los pares de variables en el *data frame*:

```
plot(iris[-5]) # se elimina de iris el factor con el tipo de flor
```



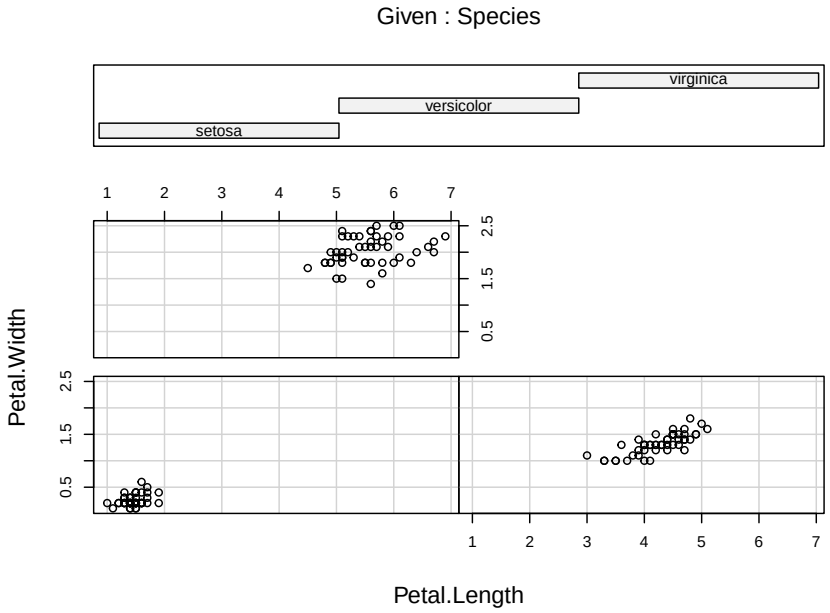
Viendo el gráfico parece que la correlación lineal entre longitud y anchura de pétalo es alta. La función `pairs` es similar a este último uso de `plot`.

```
pairs(iris[-5])
```



La función `coplot` permite generar distintos diagramas de dispersión en función de un factor:

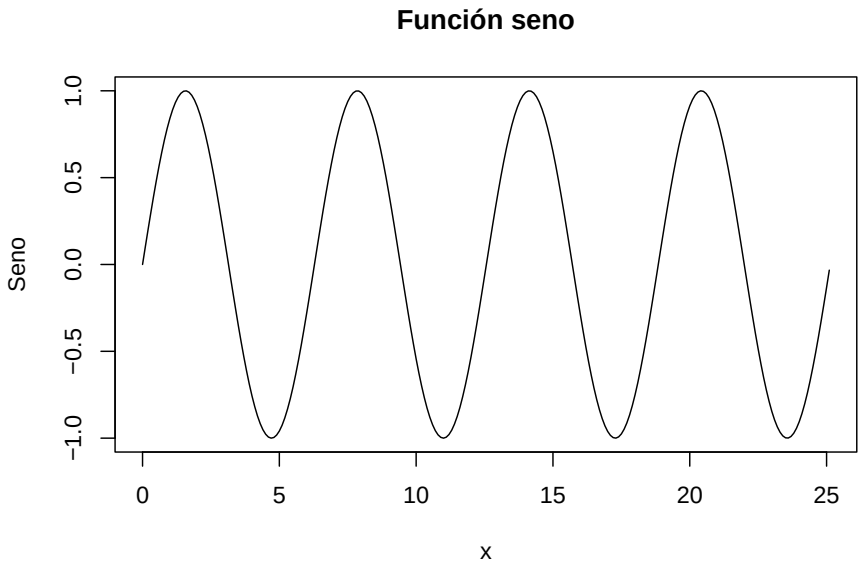
```
coplot(Petal.Width ~ Petal.Length | Species, iris)
```



12.2 Personalización de un gráfico

Existen muchas posibilidades para configurar el aspecto de un gráfico. En esta sección vamos a ver algunas. Observa este ejemplo:

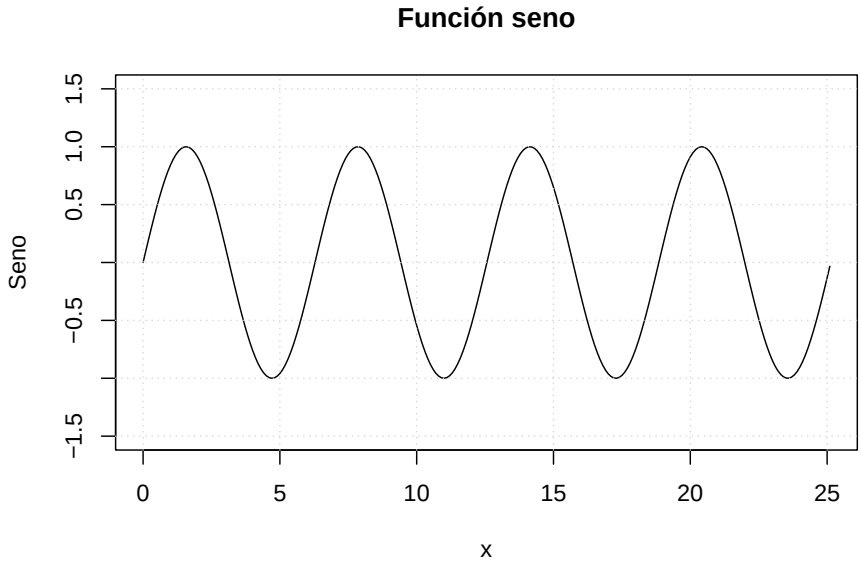
```
x <- seq(0, 8*pi, by = 0.1)
plot(x, sin(x), type = "l", xlab = "x", ylab = "Seno",
     main = "Función seno")
```



En este ejemplo, hemos usado textos para etiquetar los ejes del gráfico y hemos indicado un título. Es posible indicar el tipo de fuente y el color en que aparecen los textos.

Se puede especificar los límites de los ejes con los parámetros `xlim` e `ylim` y una malla de fondo con la función `grid`:

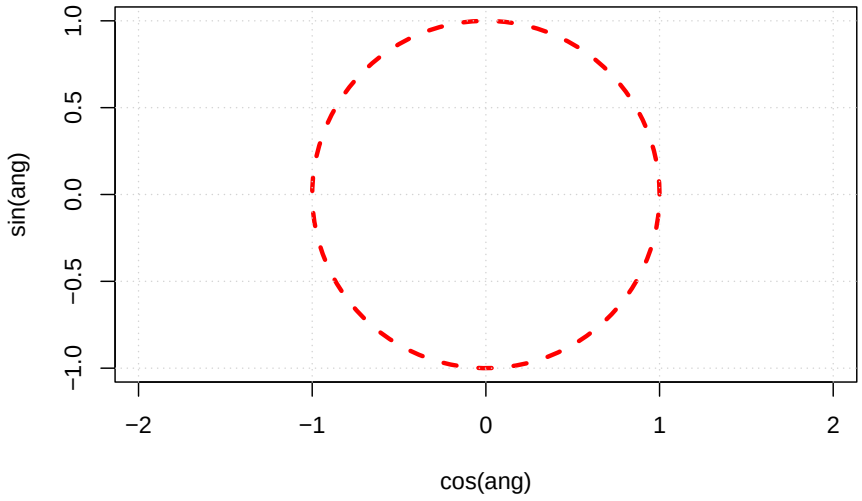
```
plot(x, sin(x), type = "l", xlab = "x", ylab = "Seno",
     main = "Función seno", ylim = c(-1.5, 1.5))
grid()
```



12.2.1 Tipos de línea, puntos y colores

Existen opciones para especificar los colores con los que se dibuja o los tipos de puntos y líneas. Veamos algún ejemplo:

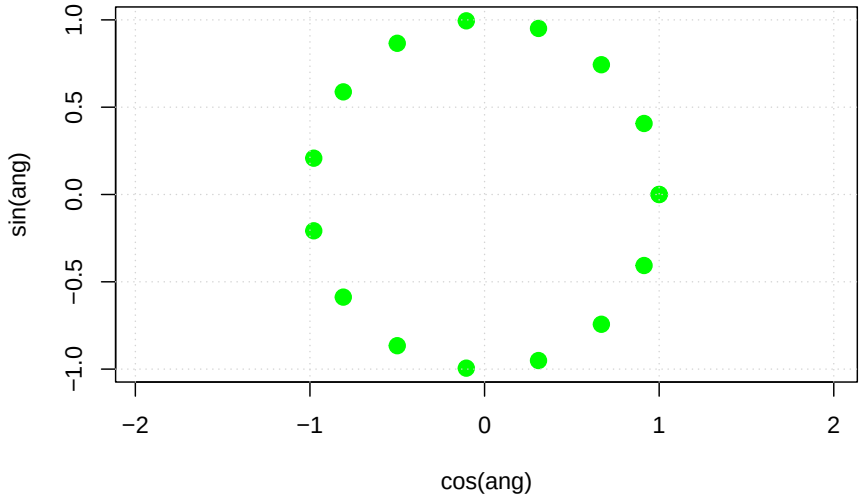
```
ang <- seq(0, 2*pi, length.out = 100)
plot(cos(ang), sin(ang), type = "l", col = "red", lty = "dashed",
     lwd = 3, asp = 1)
grid()
```



En el ejemplo se ha dibujado una circunferencia de color rojo (parámetro `col`), con líneas discontinuas (`lty` es el estilo de línea) y con una anchura de 3 unidades (parámetro `lwd`). El parámetro `asp` sirve para gestionar la proporción entre las distancias en los ejes x e y, si no se hubiera utilizado la circunferencia hubiera aparecido achatada.

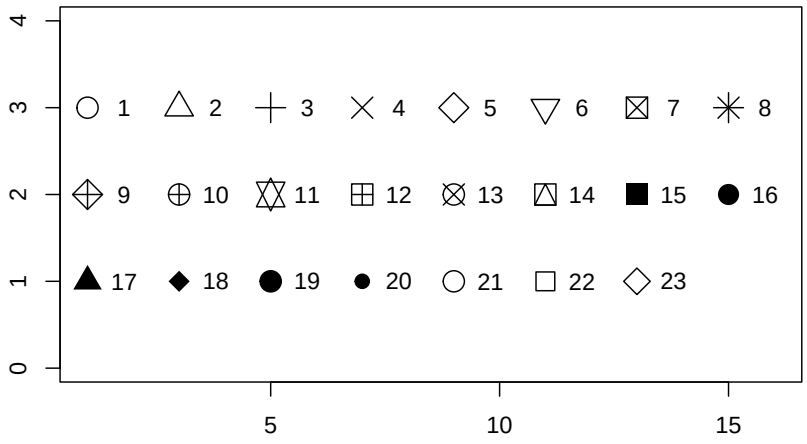
Vamos a ver otro ejemplo con el estilo de los puntos:

```
ang <- seq(0, 2*pi, length.out = 16)
plot(cos(ang), sin(ang), col = "green", pch = 19, cex = 1.5, asp = 1)
grid()
```



En este caso el parámetro `pch` indica el tipo de punto y `cex` su tamaño (1.5 veces el tamaño por defecto).

La función `colors` devuelve un vector con los colores disponibles. El siguiente gráfico muestra los distintos tipos de punto que se pueden usar con el parámetro `pch`.



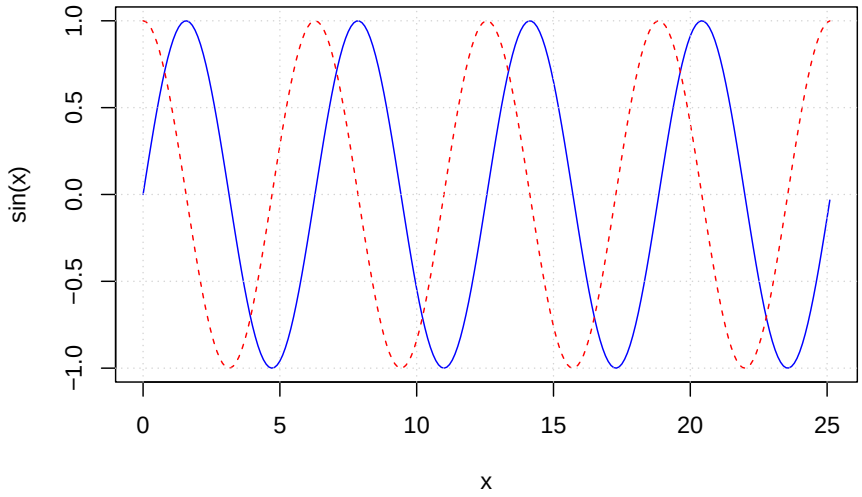
12.3 Funciones de nivel bajo

Las funciones de nivel bajo sirven para añadir información a un gráfico existente creado con una función de nivel alto, como `plot`. Vamos a estudiar alguna de ellas.

12.3.1 Funciones `points` y `lines`

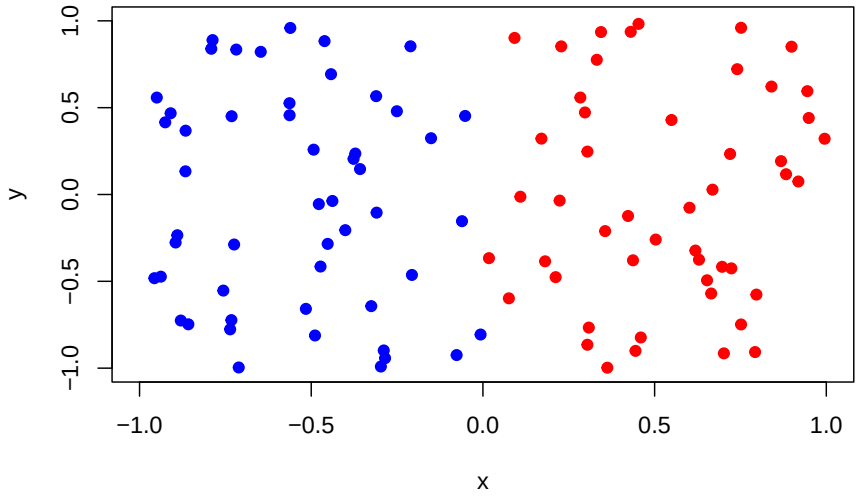
`points(x, y)` y `lines(x, y)` añaden puntos o líneas conectadas, respectivamente, al gráfico actual. En estas funciones se puede usar el argumento `type` de `plot`, siendo su valor por defecto `"p"` para `points` y `"l"` para `lines`.

```
x <- seq(0, 8*pi, by = 0.1)
plot(x, sin(x), type = "l", col = "blue")
lines(x, cos(x), col = "red", lty = "dashed") # añade líneas con coseno
grid()
```



Veamos ahora un uso de `points`. Vamos a generar puntos aleatorios en el cuadrado de esquinas (-1, -1) y (1, 1) y vamos a dibujar con distinto color a los puntos a la izquierda y derecha del eje de ordenadas:

```
x <- runif(100, min = -1, max = 1)
y <- runif(100, min = -1, max = 1)
plot(x[x <= 0], y[x <= 0], pch = 19, col = "blue", xlim = c(-1, 1),
     ylim = c(-1, 1), xlab = "x", ylab = "y")
points(x[x > 0], y[x > 0], pch = 19, col = "red")
```



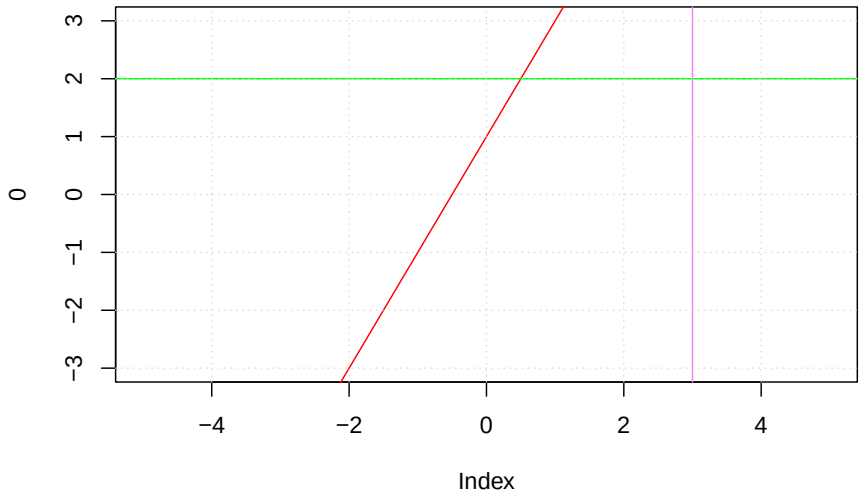
12.3.2 Función `abline`

La función `abline` sirve para añadir una línea a un gráfico. Esta función tiene varios parametros con nombre, que sirven para especificar de forma sencilla distintos tipos de líneas:

- `abline(a, b)`: línea de pendiente b y ordenada en el origen a
- `abline(h=y)`: línea horizontal
- `abline(v=x)`: línea vertical

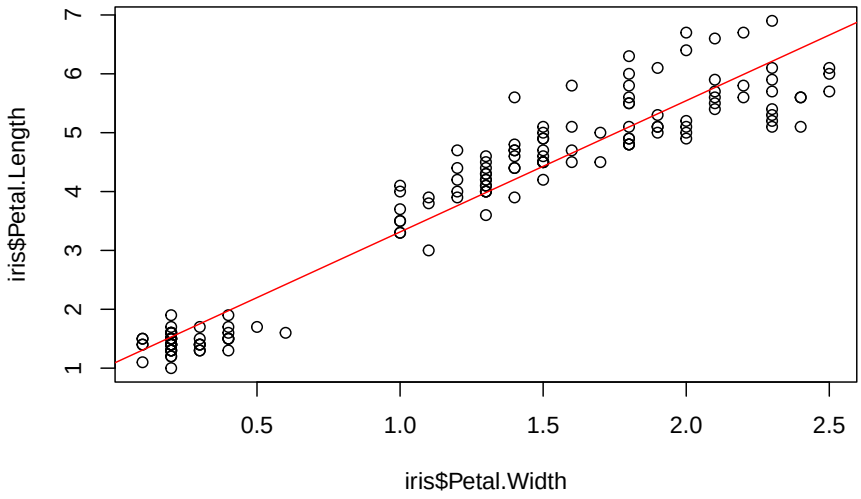
Veamos un ejemplo:

```
# El siguiente plot crea un gráfico sin dibujar nada (type = "n")
plot(0, xlim = c(-5, 5), ylim = c(-3, 3), type = "n")
abline(1, 2, col = "red")      # línea y = 2x + 1
abline(h = 2, col = "green")   # línea y = 2
abline(v = 3, col = "violet") # línea x = 3
grid()
```



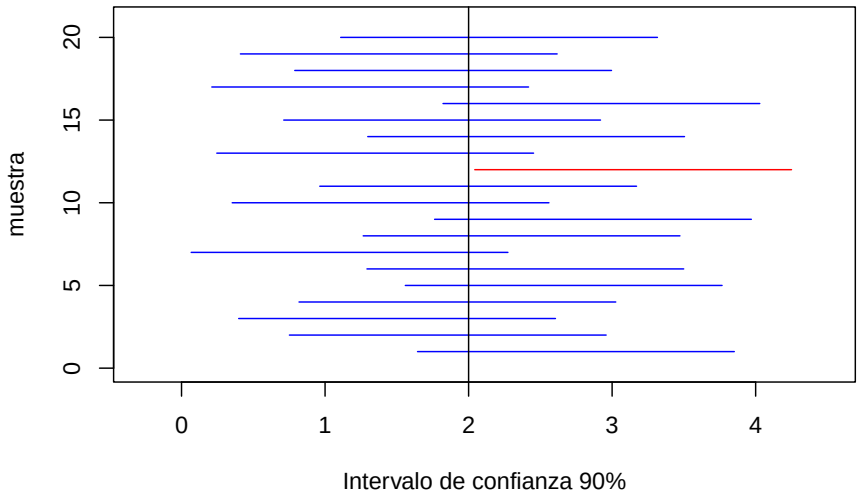
La función `abline` también permite dibujar la recta de regresión asociada a un modelo lineal:

```
plot(iris$Petal.Width, iris$Petal.Length)
modelo <- lm(Petal.Length ~ Petal.Width, data = iris)
abline(modelo, col = "red")
```



Como ejemplo vamos a generar 20 muestras de tamaño 100 de una N(2, 3) y vamos a visualizar los intervalos de confianza del 90% de la media de las muestras.

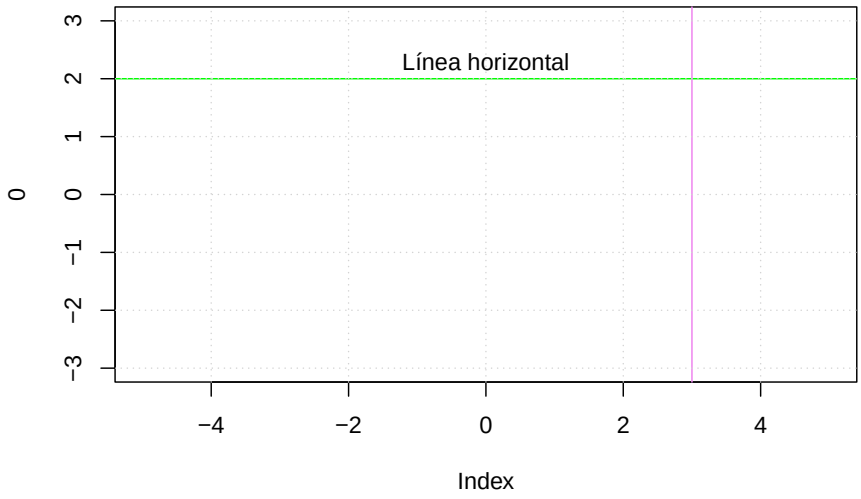
```
muestras <- matrix(rnorm(100*20, 2, 3), nrow = 20)
medias <- apply(muestras, 2, mean)
int_conf <- cbind(medias - qnorm(0.95)*3/sqrt(20),
                  medias + qnorm(0.95)*3/sqrt(20))
plot(c(min(int_conf), max(int_conf)), c(0, 21), type = "n",
     xlab = "Intervalo de confianza 90%", ylab = "muestra")
for (m in 1:20) {
  if (int_conf[m, 1] <= 2 && 2 <= int_conf[m, 2]) {
    lines(c(int_conf[m, 1],int_conf[m, 2]), c(m, m), col = "blue")
  } else {
    lines(c(int_conf[m, 1],int_conf[m, 2]), c(m, m), col = "red")
  }
}
abline(v = 2)
```



12.3.3 La función `text`

Esta función permite escribir texto en el gráfico:

```
# El siguiente plot crea un gráfico sin dibujar nada (type = "n")
plot(0, xlim = c(-5, 5), ylim = c(-3, 3), type = "n")
abline(h = 2, col = "green")  # línea y = 2
abline(v = 3, col = "violet") # línea x = 3
grid()
text(0, 2.3, "Línea horizontal")
```

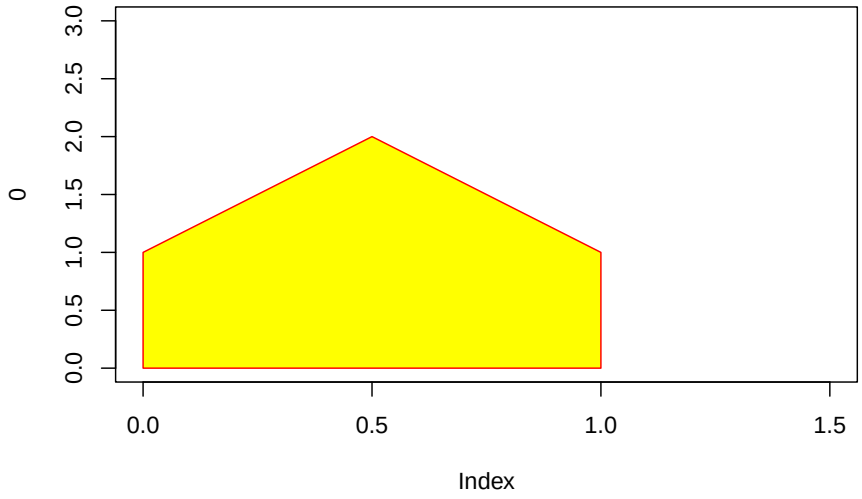


La función `text` tiene parámetros para especificar el tipo de letra, justificación del texto, tamaño, etcétera.

12.3.4 La función `polygon`

Esta función dibuja un polígono, que puede ser rellenado con color y/o líneas opcionalmente:

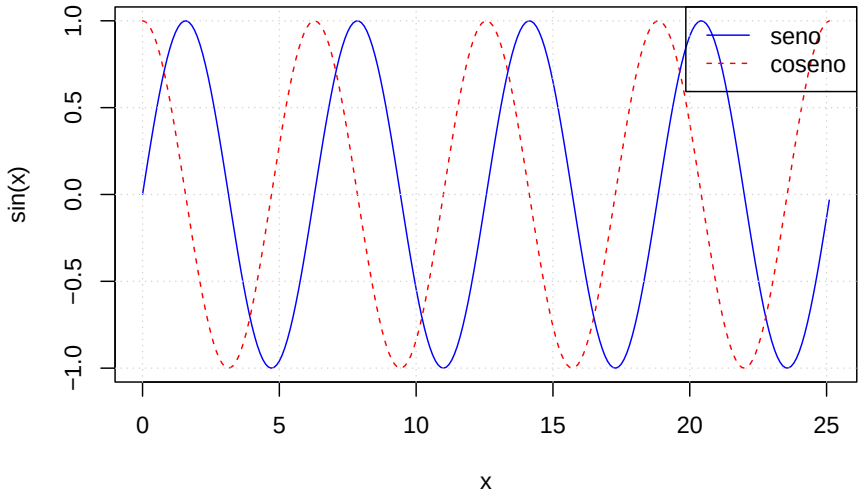
```
plot(0, xlim = c(0, 1.5), ylim = c(0, 3), type = "n")
x <- c(0, 1, 1, 0.5, 0)
y <- c(0, 0, 1, 2, 1)
polygon(x, y, col = "yellow", border = "red")
```



12.3.5 La función `legend`

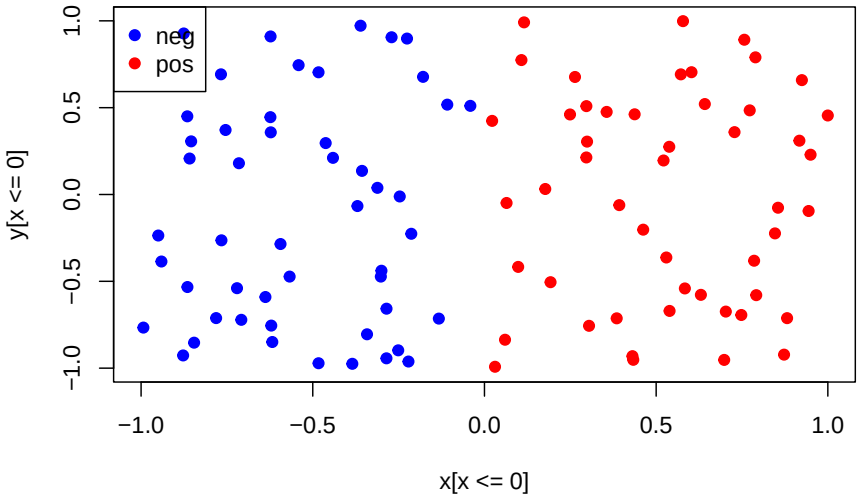
La función `legend` añade una leyenda al gráfico actual en un determinado lugar. Vamos a ver algún ejemplo:

```
x <- seq(0, 8*pi, by = 0.1)
plot(x, sin(x), type = "l", col = "blue")
lines(x, cos(x), col = "red", lty = "dashed")
grid()
legend("topright", legend = c("seno", "coseno"),
       col = c("blue", "red"), lty = c("solid", "dashed"))
```



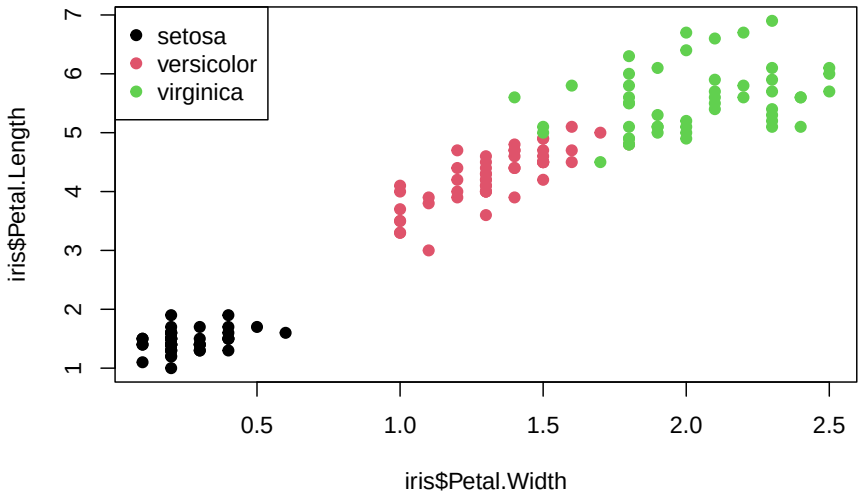
Veamos otro ejemplo con un gráfico previo:

```
x <- runif(100, min = -1, max = 1)
y <- runif(100, min = -1, max = 1)
plot(x[x <= 0], y[x <= 0], pch = 19, col = "blue", xlim = c(-1, 1),
     ylim = c(-1, 1))
points(x[x > 0], y[x > 0], pch = 19, col = "red")
legend("topleft", legend = c("neg", "pos"), col = c("blue", "red"),
       pch = c(19, 19))
```



En un último ejemplo, más avanzado, mostramos un diagrama de dispersión de la longitud de pétalo, frente a su anchura para las flores del *data frame* `iris`. Los puntos se colorean según el factor `Species`.

```
plot(iris$Petal.Width, iris$Petal.Length, col = iris$Species, pch = 19)
legend("topleft", legend = levels(iris$Species), col = 1:3, pch = 19)
```

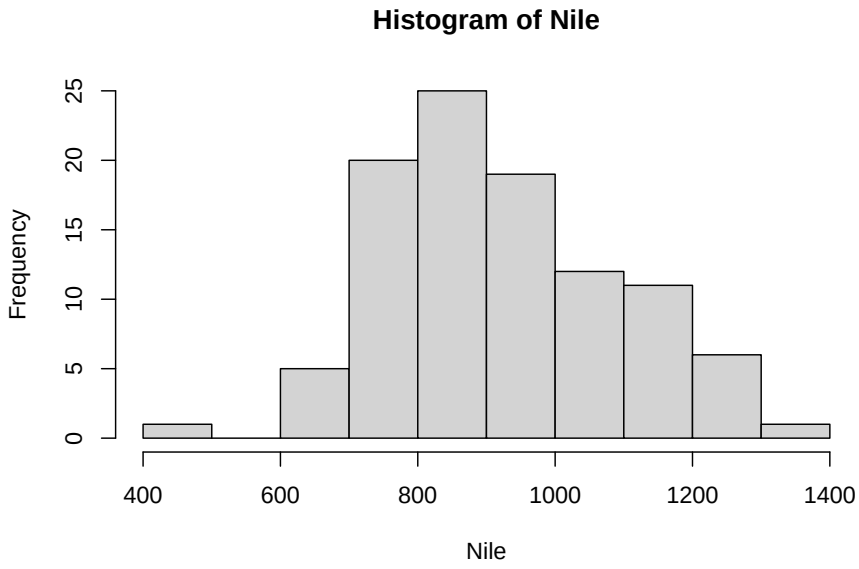


Los colores elegidos para dibujar los factores aparecen por orden en la salida de la función `palette`.

12.4 Histogramas: la función `hist`

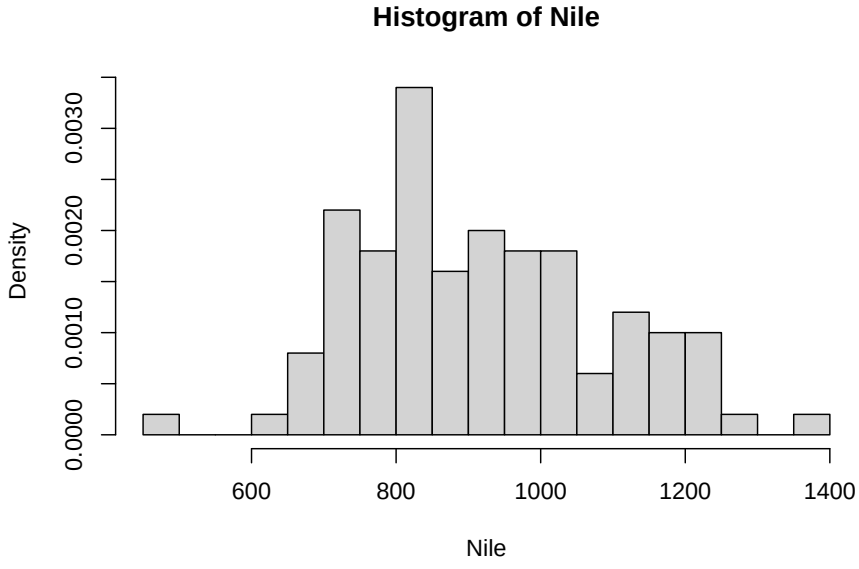
La función `hist` produce un histograma del vector numérico que recibe como parámetro. El número de clases es seleccionado automáticamente, pero puede ser especificado, así como los límites de las clases. Por defecto se muestran las frecuencias, pero se puede seleccionar ver las frecuencias relativas.

```
hist(Nile)
```



Veamos ahora un histograma con 15 clases y frecuencias relativas:

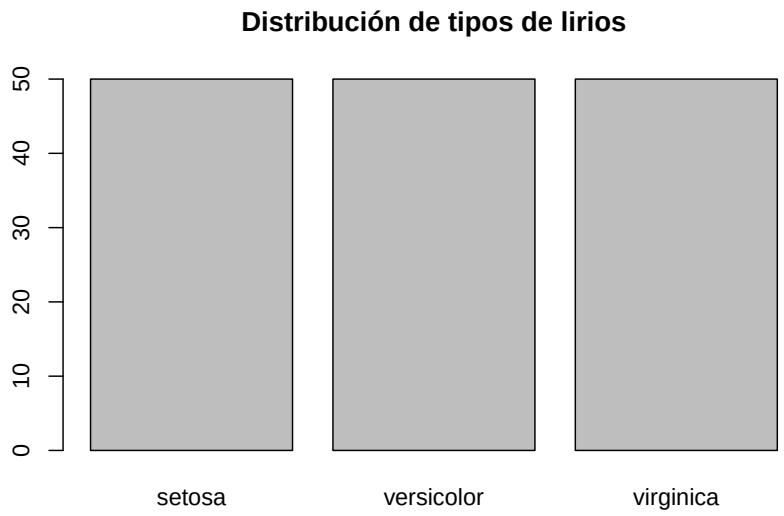
```
hist(Nile, nclass = 15, probability = TRUE)
```



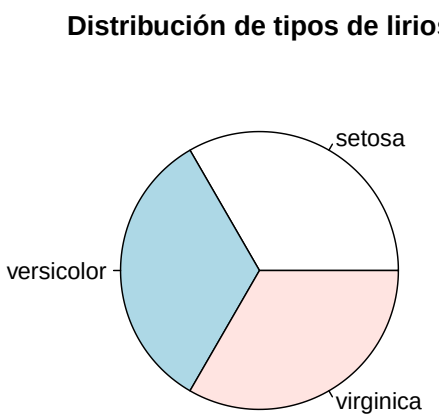
12.5 Diagramas de sectores: `pie`

Hemos visto que con `barplot` se puede obtener un gráfico de barras de los valores de una variable cualitativa. Con la función `pie` se puede obtener un diagrama de sectores.

```
barplot(table(iris$Species), main = "Distribución de tipos de lirios")
```

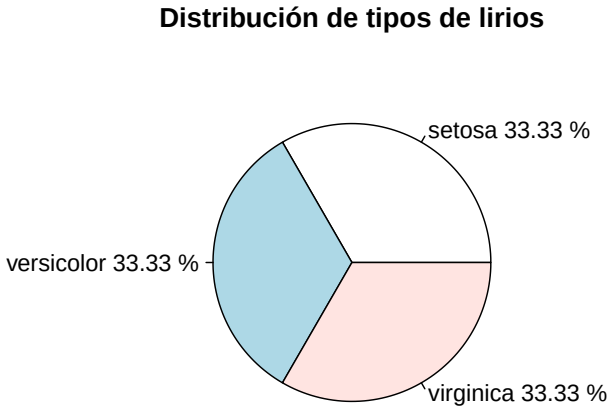



```
pie(table(iris$Species), main = "Distribución de tipos de lirios")
```



Es posible etiquetar cada sector. Por ejemplo, vamos a etiquetar cada tipo de lirio con su porcentaje de ocurrencias.

```
tabla <- prop.table(table(iris$Species))
pie(tabla,
    main = "Distribución de tipos de lirios",
    labels = paste(names(tabla), round(tabla*100, 2), "%")
)
```



12.6 Función `locator` y `par`

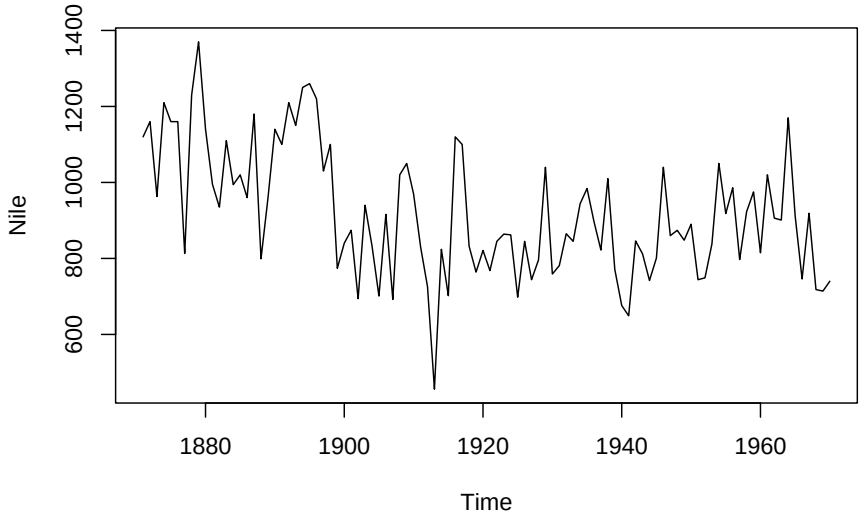
En esta sección se comentan dos funciones que pueden resultar útiles. La primera es la función `locator`. Esta función sirve para obtener las coordenadas de ciertos puntos en el gráfico. Al ejecutarla usaremos el ratón para posicionar el cursor en las coordenadas que nos interesan y pulsaremos el botón izquierdo tantas veces como coordenadas queramos obtener. Esta función resulta útil para saber aproximadamente qué coordenadas usar en una llamada a funciones como `text` o `legend` en las que podemos especificar una posición del gráfico donde ubicar un elemento. Como ejemplo, prueba lo siguiente en la consola:

```
plot(0, xlim = c(0, 2), ylim = c(0, 2), type = "n")
locator(1)
```

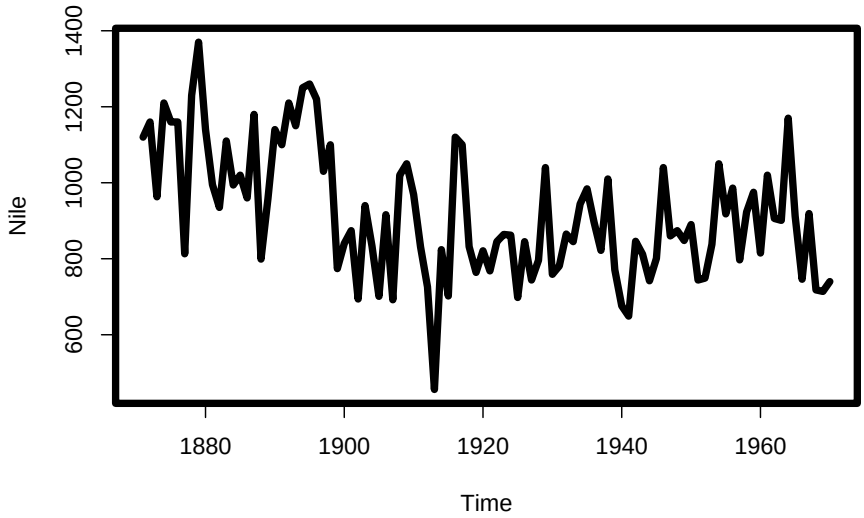
A continuación sitúate en un punto del gráfico y pulsa el botón izquierdo del ratón.

La otra función es `par`, que sirve para especificar una serie de parámetros gráficos que se usan por defecto. Por ejemplo:

```
plot(Nile)
```



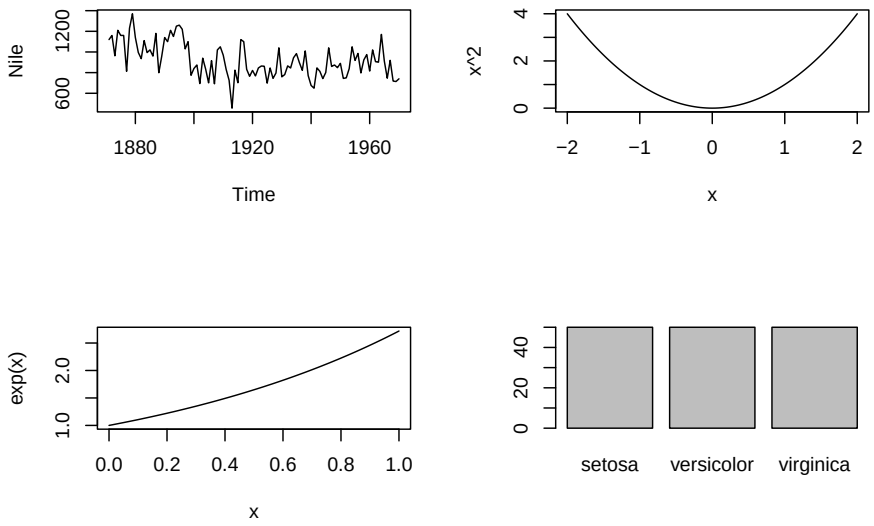

```
par(lwd = 5) # a partir de ahora el ancho de línea es 5 por defecto
plot(Nile)
```



Si se usa sin parámetros, `par` devuelve una lista con los valores actuales de

los parámetros por defecto. Un uso muy común es hacer que varios gráficos aparezcan en la misma pantalla:

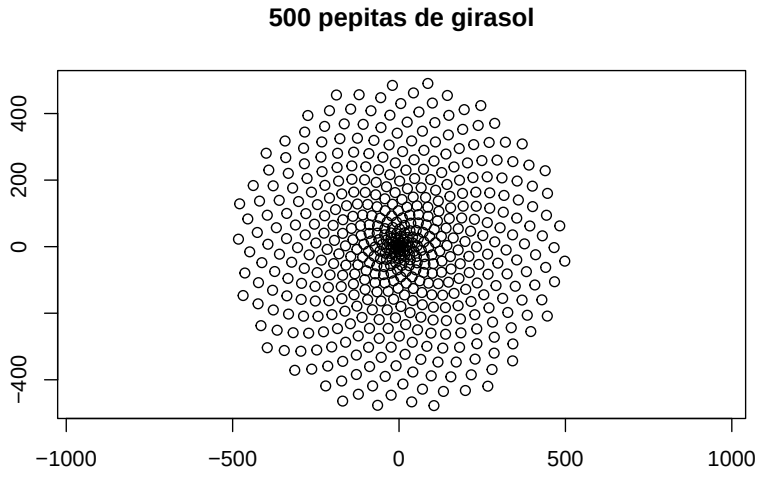
```
par(mfrow = c(2, 2))
plot(Nile)
curve(x^2, -2, 2)
curve(exp(x), 0, 1)
plot(iris$Species)
```



Si queremos restaurar el valor por defecto habrá que escribir `par(mfrow = c(1, 1))`.

12.7 Ejercicios

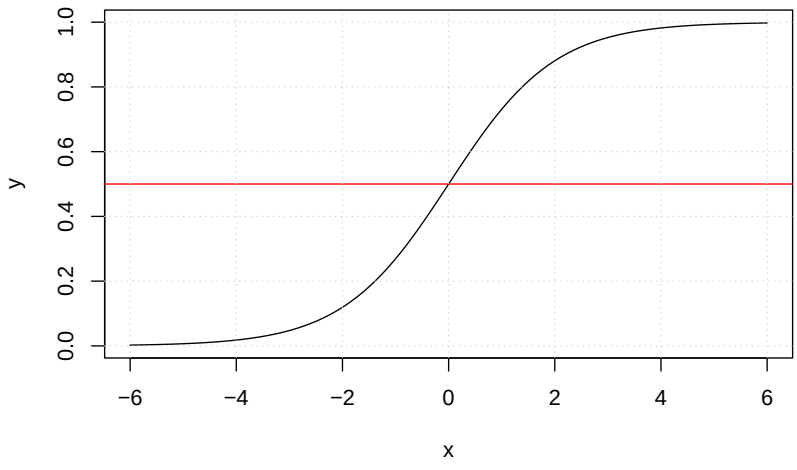
1. La disposición de las pepitas de un girasol sigue un modelo matemático. La n-ésima semilla tiene coordenadas polares $r = \sqrt{n}$ y $\alpha = \frac{137.51\pi n}{180}$. Escribe un programa que solicite el número de pepitas y las dibuje como círculos.



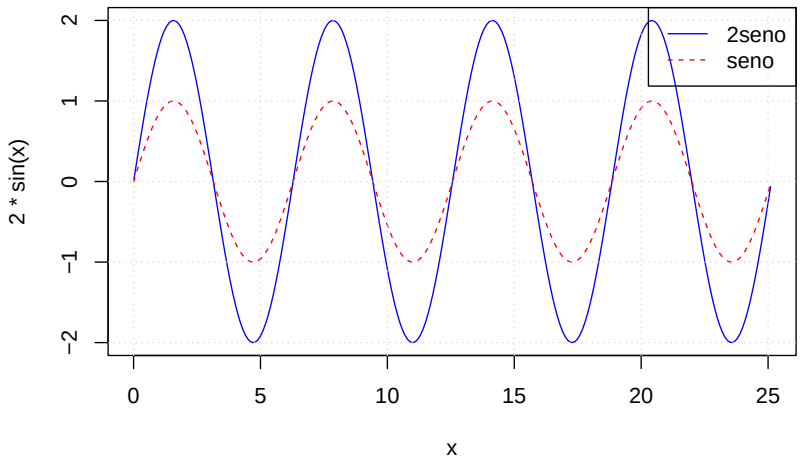
2. Representa la función logística simple:

$$P(t) = \frac{1}{1 + e^{-t}}$$

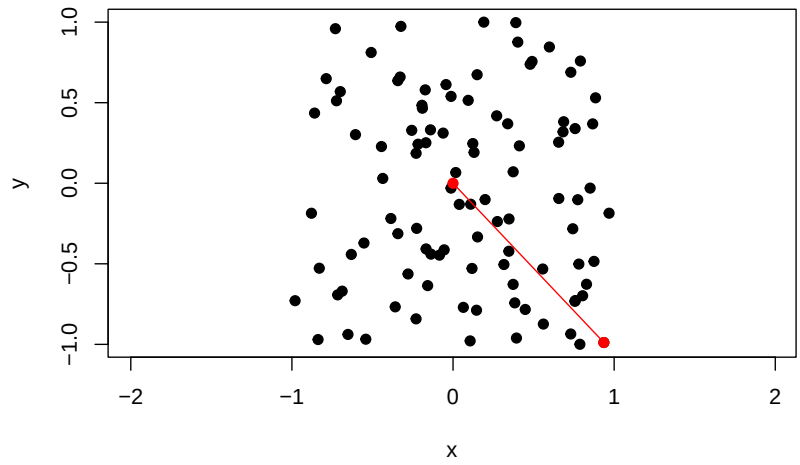
en el intervalo [-6, 6].



3. Representa las funciones $\sin(x)$ y $2\sin(x)$ en un mismo gráfico, etiquetando cada una de las funciones.



4. Genera 100 puntos aleatorios uniformemente distribuidos en el cuadrado de esquinas (-1, -1) y (1, 1). Dibuja los puntos de color negro. Dibuja de color rojo el punto más lejano al origen y une el origen y el punto con una línea roja. La distancia de un punto (x, y) al origen se calcula como $\sqrt{x^2 + y^2}$.

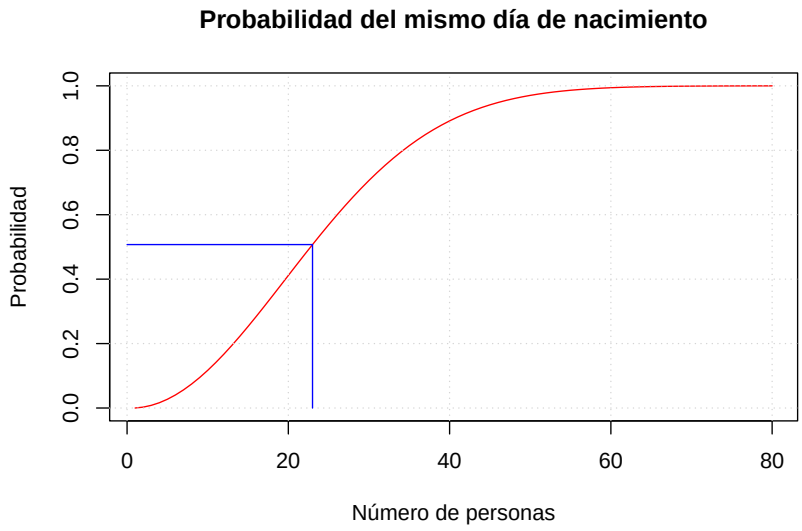


5. Supongamos que en una habitación se reunen n personas al azar, con $n \leq 365$, y ninguna de ellas ha nacido el 29 de febrero, ni hay gemelos ni

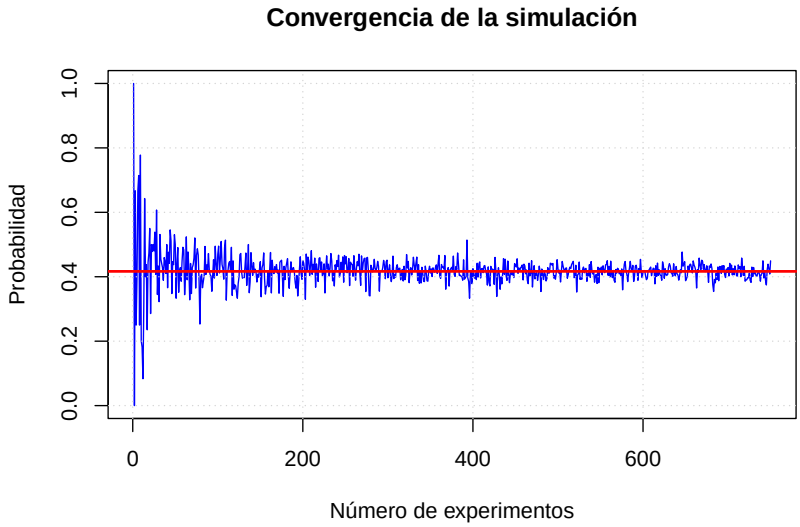
mellizos. La probabilidad de que al menos dos personas cumplan años el mismo día es:

$$1 - \frac{365}{365} \times \frac{364}{365} \times \frac{363}{365} \times ... \times \frac{365 - n + 1}{365}$$

Realiza una función que dado n devuelva la citada probabilidad. Como dato de prueba con 23 personas la probabilidad es 0.507. Haz un gráfico en el que se muestre cómo evoluciona la probabilidad en función del número de personas que haya en la habitación. Destaca en el gráfico la probabilidad para $n = 23$ (ver siguiente gráfico).



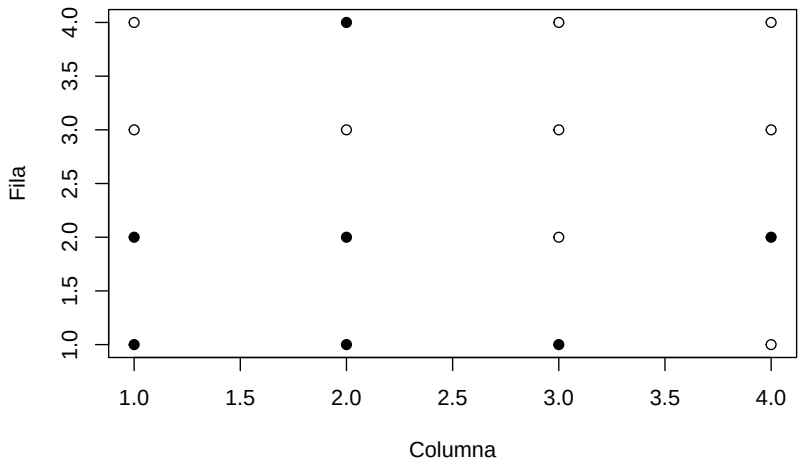
6. Supongamos que se lanzan dos dados, uno negro y otro verde, ¿Cuál es la probabilidad de que el dado negro tenga un valor mayor que el verde? En este caso la respuesta es sencilla: $\frac{15}{36}$, porque hay 36 posibles combinaciones de los valores de los dos dados y en 15 de ellas el dado negro es mayor que el verde. Sin embargo, hay situaciones en las que no es posible o fácil hacer estos cálculos. En dichas situaciones se puede recurrir a una simulación en la que se repite N veces el experimento aleatorio y se calcula la proporción de veces en la que el experimento tiene éxito (en nuestro caso el éxito equivale a que el dado negro vale más que el verde). Esa proporción es una aproximación a la probabilidad buscada. Cuanto mayor sea N mayor confianza tenemos en el valor de la aproximación. De hecho, cuando $N \to \infty$ se obtiene la solución exacta. Haz una función que calcule una aproximación a la probabilidad de que el dado negro sea mayor que el verde. Se puede escribir una función vectorizada que no use ciclos. El parámetro de la función es el valor de N. Haz una gráfica en la que se refleje cómo la simulación converge a $\frac{15}{36}$.



7. Genera una matriz aleatoria con valores de 0 y 1. Por ejemplo:

```
set.seed(10)
(m <- matrix(sample(0:1, size = 16, replace = TRUE), nrow = 4))
##      [,1] [,2] [,3] [,4]
## [1,]    0    1    0    0
## [2,]    0    0    0    0
## [3,]    1    1    0    1
## [4,]    1    1    1    0
```

Representa la matriz gráficamente con distintos símbolos para el 0 y el 1. Por ejemplo:


Fila
1.0
1.5
2.0
2.5
3.0
3.5
4.0
Columna