Introducción a R

Sesión 2
Nociones Básicas

Joaquín Ortega Sánchez

Centro de Investigación en Matemáticas, CIMAT
Guanajuato, Gto., México

Verano de Probabilidad y Estadística
Junio-Julio 2008

Outline

Outline

Introducción

Por ahora vamos a trabajar desde la cónsola

Para comenzar vamos cambiar el directorio de trabajo. Vamos a usar el directorio ”curso” para el trabajo que realizaremos en este curso. Para esto abrimos el menú *Archivo* y escogemos *Cambiar dir...*
Aparece una ventana llamada *Buscar carpeta* en la cual está la dirección del directorio actual. Si la dirección no termina con la palabra ’curso’, ubicamos la carpeta *curso*, la seleccionamos y hacemos click en el botón *Aceptar*.

Introducción

Para verificar que el cambio se efectuó, escribimos

```
> getwd()
```

y obtenemos

```
[1] "C:/Archivos de Programa/R/R-2.7.0/curso"
```

o una dirección similar terminando en el directorio `curso`.
Otra manera de cambiar el directorio es usando la instrucción `setwd()`.
Para ver que objetos se encuentran en este directorio escribimos

```
> ls()
```

y obtenemos una lista de los objetos disponibles en el directorio. El comando

```
> q()
```

cierra el programa `R`.

Outline

Sintaxis

La llamada a una función se escribe usualmente como el nombre de la función seguido por un argumento que va colocado entre paréntesis, por ejemplo

```
> data()
```

Este comando abre una ventana que tiene una lista de los conjuntos de datos disponibles. Para llamar al conjunto de datos `cars` escribimos

```
> data(cars)
```

y ahora podemos, por ejemplo, graficar los datos:

```
> plot(cars)
```

o calcular su media:

```
> mean(cars)
speed dist
15.40 42.98
```

Sintaxis

Si se escribe una expresión incompleta (por ejemplo, si se omite el segundo paréntesis al llamar una función), `R` presenta un símbolo de continuación: `+` para indicar que falta algo para completar la expresión. Por ejemplo

```
> sqrt(2
+
```

Si completamos la expresión el programa escribe el resultado:

```
> sqrt(2
+ )
[1] 1.414214
```

Es posible separar dos instrucciones con un punto y coma. Por ejemplo,

```
> 2+3; 5*4
[1] 5
[1] 20
```

Sintaxis

Si, en cambio, escribimos el nombre de la función pero omitimos los parentesis, `R` devuelve la definición de la función que a veces remite a funciones básicas. Por ejemplo

```
> log
function (x, base = exp(1))
if (missing(base)) .Internal(log(x)) else
.Internal(log(x, base))
<environment: namespace:base>
```

Sintaxis

Los operadores 'infix' son funciones con dos argumentos que usan una sintaxis especial en la cual el símbolo que representa el operador aparece entre los argumentos. Por ejemplo

```
> 17 + 25
[1] 42
> pi / 2
[1] 1.570796
```

Las operaciones aritméticas son operadores de este tipo y usan los símbolos `+` para la suma, `-` para la resta, `*` para la multiplicación, `/` para la división y `^` o `**` para la potenciación.

Sintaxis

Para expresiones simples, multiplicación y división se evalúan antes que la suma y la resta, y la potenciación las precede a todas. Por ejemplo

```
> 5+4*2
[1] 13
> (5+4)*2
[1] 18
```

Sintaxis

Observamos que en los ejemplos anteriores la respuesta incluye el índice, que es un número entero escrito entre corchetes. En cada línea de resultados de `R` se incluye un índice que corresponde al primer resultado en esa línea. Así, cuando se escriben variables con muchos resultados, es más sencillo encontrar uno de ellos en particular.

Cualquier cosa que se escriba después del símbolo `#` en la línea de comandos es un comentario y el programa lo ignora.

Sintaxis

Funciones de uso común en `R`:

Nombre	**Operación**
`abs()`	valor absoluto
`sin(), cos(), tan()`	funciones trigonométricas (en rads.)
`pi`	el número $\pi = 3{,}1415926\ldots$
`exp(), log()`	exponencial y logaritmo
`gamma()`	la función gamma
`factorial()`	la función factorial
`choose()`	número combinatorio.
`round()`	redondea el número de decimales
`cieling()`	calcula el menor entero mayor o igual q
`floor()`	calcula el mayor entero menor o igual q
`%%`	módulo
`%/%`	división entera

Tabla 1.1. Funciones numéricas.

Sintaxis

Cuando trabajamos en la consola las teclas con flechas resultan de gran utilidad:

- Flecha hacia arriba ($\uparrow$): Permite recorrer hacia atrás los comandos utilizados anteriormente.
- Flecha hacia abajo ($\downarrow$): Permite avanzar cuando se revisan los comandos utilizados anteriormente.
- Flechas laterales ($\leftarrow$, $\rightarrow$): Permiten recorrer el comando en pantalla en sentido izquierdo o derecho, respectivamente.

Sintaxis

Como ejemplo, supongamos que queremos calcular la raíz cuadrada de 5.5 pero cometemos un error al escribir sqrt:

```
> sqt(5.25)
Error: couldn't find function "sqt"
```

Usando la flecha hacia arriba en la consola recuperamos el comando errado y lo corregimos, sin necesidad de volver a escribir toda la línea:

```
> sqrt(5.25)
[1] 2.291288
```

Sintaxis

Uno de los operadores de uso más frecuente es el de asignación, que asocia nombres con valores, y se representa por los símbolos = o <- (conjunción de 'menor que' < y 'menos' -). Por ejemplo, para asignarle el valor 10 a la variable a escribimos

```
> a <- 10
```

Para ver el valor de una variable podemos escribir el nombre de la variable o usar la función print

```
> a
[1] 10
> print(a)
[1] 10
```

Sintaxis

De manera similar podemos crear una variable `b` con valor `-5` usando la instrucción

```
> b = -5
```

También es posible hacer la asignación con la flecha apuntando en sentido contrario

```
> -5 -> b
```

Cuando hacemos una asignación, `R` no escribe el valor de la asignación. Para verlo es necesario escribir el nombre de la variable, o escribir la instrucción de asignación entre paréntesis:

```
> (a <- 5)
[1] 5
```

Sintaxis

Todas las asignaciones permanecen hasta que sean reemplazadas o eliminadas. El comando `rm` se usa para eliminar explícitamente uno o varios objetos. Por ejemplo, `rm(a,b)` elimina las variables `a` y `b`. En el siguiente ejemplo cambiamos el valor de `a` y luego la eliminamos.

```
> a <- 66
> a
[1] 66
> rm(a)
> a
Error: Object "a" not found
```

Sintaxis

`R` guarda automáticamente las asignaciones de valores a variables, de modo que las variables pasan a ser objetos del programa y permanecen en el directorio de trabajo a menos que sean eliminadas.

Para borrar todos los objetos en la memoria escribimos

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

Sintaxis

Al cerrar el programa aparece un mensaje preguntando si se desea guardar una 'imagen del espacio de trabajo' (*workspace image*). Si se hace 'click' en 'Si' se guardan todos los objetos que están en el directorio de trabajo: los que estaban allí al inicio más los que se hayan creado (y no se hayan eliminado) durante la sesión. Estos objetos se guardan por defecto en el archivo '.RData' del directorio que se está usando. Si abrimos una nueva sesión desde ese directorio, todos los objetos que hayan sido guardados estarán disponibles. Si abrimos la sesión desde otro directorio, podemos cambiar de directorio usando la función `setwd()` y luego cargar los objetos guardados con el comando `load()` escribiendo `load(".RData")`.

Sintaxis

Para ver los objetos existentes es posible usar las funciones `ls()` u `objects()`, que producen una lista de los objetos disponibles en el directorio de trabajo. Si sólo deseamos listar los objetos que tienen un carácter determinado en su nombre, es posible usar la opción `pattern`

```
> ls(pat = 'm')
```

Para restringir la búsqueda a objetos cuyos nombres inicien con esta letra escribimos

```
> ls(pat = '^m')
```

La función `ls.str` muestra algunos detalles sobre los objetos en la memoria.

Outline

Ayuda

Una de las ventajas de `R` es la extensa documentación disponible en línea sobre funciones y conjuntos de datos. El comando `help()` abre una ventana que describe el uso de la función que aparezca como argumento, o las características del conjunto de datos, si es el caso. Por ejemplo

```
> help(log)
```

abre una ventana que describe la sintaxis de las distintas funciones de R que calculan logaritmos, su estructura, la de las exponenciales (por ser las funciones inversas) y presenta referencias y ejemplos.
Otra manera de escribir esta instrucción es

```
> ?log
```

Ayuda

Los comandos `help(log)` y `?log` producen el mismo resultado. Sin embargo, para obtener ayuda sobre caracteres no convencionales es necesario usar el último formato:
`> ?'*'.`
Por defecto, la función `help` sólo busca en los paquetes que están cargados en memoria. La opción `try.all.packages`, que por defecto toma el valor `FALSE`, permite buscar en todos los paquetes si su valor es `TRUE`:

```
> help('truehist')
No documentation for 'truehist' in specified
    packages and libraries: you could try
    'help.search("truehist")'
```

Ayuda

```
> help('truehist',try.all.packages=TRUE)
Help for topic 'truehist' is not in any loaded
    package but can be found in the following
    packages:

    Package Library
    MASS C:/PROGRA 1/R/R-24 1.0/library
> help('truehist', package='MASS')
```

Ayuda

Otras funciones que resultan útiles al buscar funciones que realicen determinadas tareas son `apropos()` y `help.search()`. Veamos ejemplos de su uso:

```
> apropos("sort")
 [1] "sortedXyData" "is.unsorted" "sort"
"sort.default" "sort.int"
 [6] "sort.list" "sort.POSIXlt"
```

El resultado es una lista de las funciones cuyo nombre incluye los caracteres 'sort'. En cambio

```
> help.search("sort")
```

abre una ventana donde aparecer una lista de todas las funciones que o tienen `sort` como alias o ésta palabra aparece en su titulo.

Ayuda

Una manera alternativa de obtener ayuda es ejecutar el comando

```
> help.start()
updating HTML package listing
updating HTML search index
If nothing happens, you should open
'C:\ARCHIV~1\R\R-24~1.0\doc\
    html\index.html' yourself
```

Esta instrucción abre una ventana del navegador de uso habitual en la computadora y aparece una página de ayuda de `R` que tiene ligas a los manuales, recursos, información sobre paquetes, etc. Especialmente útil es la liga a `Search Engine & Keywords` que abre un motor de busqueda.

Ayuda

Otra función de interés es `example()`, que corre los ejemplos disponibles de la función que aparezca como argumento.
Por ejemplo,

```
> example(image)
```

produce una serie de cuatro gráficas en una misma ventana.
Para que aparezca la siguiente gráfica es necesario apretar la tecla Intro.

Outline

Lectura de datos: Leer un archivo

Vamos a leer una tabla guardada en un archivo de texto usando la función `read.table()`. En el directorio `curso` hay un archivo llamado `MEXpob.txt` que tiene una tabla que muestra la población (en millones) de México desde 1895. El siguiente comando lee el archivo y lo guarda con el nombre `MEXpob`:

```
> MEXpob <- read.table("MEXpob.txt",
    header=TRUE)
> MEXpob
    Año Pob.
1  1895 12.6
2  1900 13.6
3  1910 15.2
4  1921 14.3
⋮   ⋮    ⋮
```

Lectura de datos: Leer un archivo

La opción `header = T` usa los nombres de las variables que aparecen en la primera línea del archivo. Ahora hacemos un gráfico de estos datos usando la instrucción

```
> plot(Pob. ~ Año, data=MEXpob, pch=16)
```

La opción `pch=16` hace que el programa utilice puntos negros para realizar la gráfica.
Una opción que resulta cómoda es usar la función `file.choose()` para buscar el archivo de manera interactiva. La instrucción

```
> prueba <- read.table(file.choose())
```

abre una ventana que permite buscar en el directorio el lugar en que se encuentra el archivo que deseamos abrir, sin necesidad de saber su nombre exacto o su ubicación.

Lectura de datos: Usar la linea de comandos

Los datos de la tabla 1.2 se refieren a una banda elástica y dan la distancia recorrida por la banda al lanzarla usando el extremo de una regla, en función del estiramiento de la banda.

Estiramiento	Distancia
46	148
54	182
48	173
50	166
44	109
42	141
52	166

Tabla 1.2: Distancia (cm) recorrida por una banda elástica en función del estiramiento (cm).

Lectura de datos: Usar la linea de comandos

Vamos a crear un cuadro de datos o 'data frame' con estos datos desde la línea de comandos. Los cuadros de datos son estructuras de forma matricial que permiten combinar datos de diversos tipos: numéricos, lógicos, caracteres, etc. Llamaremos `bandaelastica` al cuadro de datos que crearemos. Las instrucciones son:

```
> bandaelastica <- data.frame(
estiramiento=c(46,54,48,50,44,42,52),
distancia=c(148,182,173,166,109,141,166))
```

Lectura de datos: Usar la linea de comandos

```
> bandaelastica
 estiramiento distancia
1           46       148
2           54       182
3           48       173
4           50       166
5           44       109
6           42       141
7           52       166
```

Lectura de datos: La función `scan`

Otra manera de crear archivos desde la línea de comandos es con la función `scan()`, que permite introducir datos desde el teclado o leerlos desde un archivo. Los datos que van a ser introducidos deben estar separados por un espacio en blanco, un espacio de tabulación o deben estar en una línea nueva. Si se usa la función sin argumento: `scan()` los datos deben introducirse usando el teclado. Para indicarle al programa que hemos finalizado, hay que dejar una línea en blanco. Veamos un ejemplo:

```
w <- scan()
1: 1 2 3 4 5
6: 6 7 8 9 10
11:
Read 10 items
> w
[1] 1 2 3 4 5 6 7 8 9 10
```

Lectura de datos: La función `scan`

La función `scan` espera por defecto que las componentes del vector sean números. Si se desea que sean caracteres puede indicarse usando la opción `what`. Para introducir caracteres escribimos `what= " "`. También es posible especificar un separador distinto con la opción `sep`. Veamos un ejemplo:

```
> nombres <- scan(what=" ", sep="-")
1: Carlos-Juan-Jose Antonio
4:
Read 3 items
> nombres
[1] "Carlos" "Juan" "Jose Antonio"
```

Observe que con la opción sep asignada como lo hemos hecho en el ejemplo, el espacio en el nombre `Jose Antonio` se lee correctamente.

Lectura de datos: La función `scan`

Adicionalmente, la función `scan()` puede usarse para leer datos almacenados en formato ASCII. Para esto, basta con poner el nombre del archivo y su ubicación, entre comillas, en el argumento de la función. También es posible usar la función `file.choose` que abre una ventana que permite buscar el archivo en el directorio.

Es posible usar unar interface similar a una hoja de cálculo para modificar un archivo. Por ejemplo, si escribimos

```
> data.entry(bandaelastica)
```

se abre una ventana en la cual podemos hacer modificaciones al conjunto de datos.

Lectura de datos

Observación

- Siempre es conveniente ver el archivo de datos antes de leerlo desde `R` para estar seguro de que tiene la información que queremos y que su formato es apropiado.
- Para representar directorios y subdirectorios, use la barra inclinada hacia delante / y no la barra inclinada hacia atrás \.
- Si su computadora está conectada a internet, es posible leer directamente un archivo de datos en formato de texto usando la función `scan()`. La sintaxis básica es

```
> dat <- scan("http://www..."')
```

- Si el directorio o el nombre del archivo tiene espacios, es necesario incluir una barra inclinada hacia atrás (\) antes del espacio.

Ejercicio

Ejercicio 1

1. Lea el archivo de datos equipaje.txt que se encuentra en el directorio de trabajo y guardelo con el nombre `equipaje`.
2. Vea su estructura con el comando `str`. Liste todas las componentes.
3. Abralo con el editor de textos y cambie el dato `9.5` por `9.3`.
4. Añada las siguientes componentes al final del archivo: `7.8, 6.9, 8.2`
5. Cierre al archivo.

Outline

Ventanas Script

Ahora vamos a trabajar con un `script`

La consola es útil para comandos sencillos que van a ser ejecutados de inmediato pero en general resulta más conveniente usar las ventanas `script`, que permiten escribir instrucciones sin que se ejecuten de manera automática, y de esta manera permite desarrollar un guión o 'script' constituido por una serie de instrucciones concatenadas. Estas ventanas pueden ser guardadas para ser ejecutadas o modificadas posteriormente.

Ventanas Script

Para abrir una ventana de este tipo abrimos el menú 'File' y seleccionamos 'New script'. Las instrucciones que se escriben sobre esta nueva ventana no se ejecutan automaticamente al presionar la tecla 'Intro': Hay que seleccionar la instrucción y luego presionar 'Control-r' o bien presionar el botón derecho del ratón y seleccionar 'Run line or selection'. Las instrucciones seleccionadas se ejecutan en la consola y los resultados aparecen allí.

Ventanas Script

Las instrucciones que vamos a ejecutar a continuación se encuentran en un guión llamado 'Verano2'. Para abrirlo vamos al menú *Archivo* y seleccionamos *Abrir script* y en la ventana que aparece seleccionamos *Verano2.R*.

c

La función `c`
Esta función permite combinar varios elementos para crear un vector. Basta listar los elementos en el orden deseado, separando cada elemento por una coma:

```
> (x <- c(1, 1.5, 2, 2.5))
[1] 1.0 1.5 2.0 2.5
```

La función c también puede usarse con variables. Por ejemplo, si olvidamos incluir el número `3` como último elemento en el vector, podemos añadirlo de la siguiente manera:

```
> (x <- c(x,3))
[1] 1.0 1.5 2.0 2.5 3.0
```

c

También puede usarse con caracteres:

```
(y <- c('esto', 'es', 'un', 'ejemplo'))
[1] "esto" "es" "un" "ejemplo"
```

y con combinaciones de números y caracteres, aunque en este caso todos los elementos serán considerados como caracteres (más adelante explicaremos esto) y en consecuencia no pueden realizarse operaciones aritméticas entre ellos.

```
(z <- c(x,'a'))
[1] "1" "1.5" "2" "2.5" "3" "a"
```

seq

La función `seq`

Esta función permite formar sucesiones regulares de números.
Su sintaxis es
`seq(lím. inferior, lím. superior, incremento).`
Veamos algunos ejemplos:

```
> seq(0,100,5)
[1]  0  5 10 15 20 25 30 35 40 45 50 55 60
[14] 65 70 75 80 85 90 95 100
> seq(1955,1966,1)
[1] 1955 1956 1957 1958 1959 1960 1961 1962
[9] 1963 1964 1965 1966
> seq(10,12,0.2)
[1] 10.0 10.2 10.4 10.6 10.8 11.0 11.2 11.4
[9] 11.6 11.8 12.0
```

seq

Si no se especifica el incremento, el programa asume el valor 1. Si no se especifica el límite inferior, el programa también asume el valor 1.

```
> seq(1955,1966)
[1] 1955 1956 1957 1958 1959 1960 1961 1962
[9] 1963 1964 1965 1966
> seq(5)
[1] 1 2 3 4 5
```

La expresión para la función `seq` puede abreviarse cuando el incremento toma el valor 1 usando dos puntos entre el límite inferior y el límite superior:

```
> (b <- 1:10)
[1] 1 2 3 4 5 6 7 8 9 10
```

seq

La limitación del valor del incremento puede remediarse usando operaciones aritméticas sobre el vector. Por ejemplo,

```
> 50:60 /5
[1] 10.0 10.2 10.4 10.6 10.8 11.0 11.2 11.4
[9] 11.6 11.8 12.0
```

y obtenemos el mismo resultado que en uno de nuestros ejemplos anteriores. También es posible construir sucesiones en orden decreciente, usando incrementos negativos y colocando los extremos en orden inverso:

```
> seq(10,1,-1)
[1] 10  9  8  7  6  5  4  3  2  1
> 10 : 1
[1] 10  9  8  7  6  5  4  3  2  1
```

rep

La función `rep`

Esta función sirve para repetir un patrón dado. Su sintaxis es `rep(patrón, número de repeticiones)`. Veamos algunos ejemplos:

```
> rep(10,3)
[1] 10 10 10
> rep(c(0,5), 4)
[1] 0 5 0 5 0 5 0 5
> rep(c('se','va'),3)
[1] "se" "va" "se" "va" "se" "va"
> rep(1:5,2)
[1] 1 2 3 4 5 1 2 3 4 5
```

rep

El número de repeticiones puede ser un vector, en cuyo caso debe tener el mismo número de componentes que el patrón, y entonces cada elemento es repetido el número de veces correspondiente.

```
> rep(c(10,20),c(2,4))
[1] 10 10 20 20 20 20
> rep(1:3,1:3)
[1] 1 2 2 3 3 3
```

La función `rep` puede usarse para construir alguno de los vectores:

```
> rep(1:3,rep(4,3))
[1] 1 1 1 1 2 2 2 2 3 3 3 3
```

rep

Si sabemos la longitud del vector que queremos obtener en lugar del número de veces que el patrón debe ser repetido, podemos usar la opción `length` en lugar del número de repeticiones. Por ejemplo, si deseamos construir un vector de longitud 10 usando el patrón 1 2 3 4 escribimos

```
> rep(c(1,2,3,4), length=10)
[1] 1 2 3 4 1 2 3 4 1 2
```

El Operador Indice

El operador índice

Este operador sirve para extraer subconjuntos de objetos de datos en `R`. La sintaxis es `objeto[índice]` donde `objeto` puede ser cualquier objeto de datos en `R`, e `índice` puede ser alguna de las siguientes alternativas:

- Enteros positivos que corresponden a la posición que ocupan los datos buscados en el objeto. Por ejemplo, el conjunto de datos 'letters' contiene las 26 letras minúsculas que se usan en inglés. Para seleccionar la quinta letra escribimos

```
> letters[5]
[1] "e"
> letters[c(2,8,16)]
[1] "b" "h" "p"
```

El Operador Indice

- Enteros negativos que corresponden a la posición de los datos que deben ser excuidos

```
> letters [-5]
[1] "a" "b" "c" "d" "f" "g" "h" "i" "j"
[10] "k" "l" "m" "n" "o" "p" "q" "r" "s"
[19] "t" "u" "v" "w" "x" "y" "z"
> letters [-c(2,8,16)]
[1] "a" "c" "d" "e" "f" "g" "i" "j" "k"
[10] "l" "m" "n" "o" "q" "r" "s" "t" "u"
[19]"v" "w" "x" "y" "z"
```

El Operador Indice

- Valores lógicos; valores verdaderos corresponden a los puntos que deseamos incluir, valores falsos a los que deseamos excluir.

  Ejemplo:

  1. Asignamos a la variable `a` los enteros del 1 al 26 por la asignación `a <- 1:26`.
  2. Luego verificamos que la expresión `a < 13` produce una sucesión de `'T'` y `'F'` en la cual `'T'` aparece en los primeros 12 lugares y `'F'` en el resto.
  3. Finalmente, usamos la expresión `a < 13` como índice para obtener el subconjunto de las 12 primeras letras.

El Operador Indice

```
a <- 1:26
> a
[1] 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17
[18] 18 19 20 21 22 23 24 25 26
> a < 13
[1] T T T T T T T T T T T T F F F F F F F F F
[22] F F F F F
> letters [a<13]
[1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j"
[11] "k" "l"
```

El Operador Indice

Para utilizar la conjunción de dos expresiones lógicas es necesario usar el operador `&`. Por ejemplo, para especificar una desigualdad del tipo $a < x < b$, es necesario escribir esta desigualdad como la conjunción de las desigualdades `a<x` y `x<b`.

Así, para escoger las letras con índices que satisfagan `5<x<10`, escribimos

```
> letters[5 < a & a < 10]
[1] 'f' 'g' 'h' 'i'
```

El Operador Indice

A continuación presentamos una lista de los operadores lógicos y su expresión en R.

Símbolo	Relación
$<$	Menor que
$>$	Mayor que
$<=$	Menor o igual que
$>=$	Mayor o igual que
$==$	Igual a
$!=$	No es igual a

Tabla 1.3. Operadores lógicos.

El Operador Indice

Los operadores lógicos pueden usarse, por ejemplo, para seleccionar los valores pares de una variable con valores enteros.

```
> (x <- rbinom(20, 40, prob=1/3))
[1] 12 19 18 11 8 15 14 11 14 12 13 18
[12] 20 15 11 7 17 13 15 14
> x[x%% 2==0]
[1] 12 18 8 14 14 12 18 20 14
```

El sistema de indexación permite usar valores lógicos, que son reciclados si la longitud del vector es menor que la del vector de los valores lógicos. La siguiente instrucción permite obtener los valores que ocupan los lugares impares

```
> x[c(T,F)]
[1] 12 18 8 14 14 13 20 11 17 15
```

which

La función which

Para hacer una lista de los índices de las observaciones que satisfacen cierta condición es posible usar la función which

```
> which(trees$Volume > 50)
[1] 26 27 28 29 30 31
> trees[which(trees$Volume > 50),]
    Girth  Height  Volume
26   17.3      81    55.4
27   17.5      82    55.7
28   17.9      80    58.3
29   18.0      80    51.5
30   18.0      80    51.0
31   20.6      87    77.0
```

expand.grid

La función `expand.grid`

Esta función crea un cuadro de datos con todas las combinaciones posibles de vectores o factores que aparecen como argumentos de la función. Veamos un ejemplo:

```
> expand.grid(edad=c(20,40),peso=c(50,70),
    sexo=c('Masculino','Femenino'))
```

	edad	peso	sexo
1	20	50	Masculino
2	40	50	Masculino
3	20	70	Masculino
4	40	70	Masculino
5	20	50	Femenino
6	40	50	Femenino
7	20	70	Femenino
8	40	70	Femenino

Ejercicio

Ejercicio 2

1. Usando las instrucciones `seq` y `rep` genere las siguientes sucesiones:
   10 10 10 10 10 9 9 9 9 8 8 8 7 7 6 5 4 4 3 3 3 2 2 2 2 1 1 1 1 1

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

   100.0000 100.2222 100.4444 100.6667 100.8889 101.1111 101.3333 101.5556 101.7778 102.0000
2. Usando el conjunto de datos `equipaje` que creó en el ejercicio anterior, obtenga todos los elementos que estan entre 5 y 7 kilos.

Muestras Aleatorias

La función `sample` permite generar muestras aleatorias de un conjunto dado. La sintaxis es

```
sample(x, size, replace = FALSE, prob = NULL)
```

donde

- `x` es el conjunto a partir del cual queremos obtener la muestra,
- `size` es el tamaño de la muestra,
- `replace` permite indicar si se permiten repeticiones (`replace = TRUE`) o no y finalmente
- `prob` es un vector de probabilidades si se desea hacer un muestreo pesado y no uniforme. Veamos un ejemplo.

Muestras Aleatorias

```
> xy <- c('malo','regular','bueno')
> sample(xy,10,replace=T)
[1] "malo" "bueno" "bueno" "bueno" "regular"
[6] "bueno" "malo" "regular" "regular"
[10] "bueno"
```

Ahora hacemos un muestreo no-uniforme especificando el vector de probabilidades para cada valor.

```
> pp <- c(0.1,0.1,0.8)
> sample(xy,10,replace=T,prob=pp)
[1] "bueno" "bueno" "bueno" "bueno" "bueno"
[10] "bueno" "bueno" "bueno" "bueno" "malo"
```

Muestras Aleatorias

`R` también tiene funciones para obtener muestras a partir de numerosas distribuciones de probabilidad. La sintaxis común de estas funciones es `r`*dist*, donde *dist* designa la distribución; por ejemplo, para generar valores a partir de la distribución normal usamos `rnorm`. Según la distribución, puede ser necesario especificar uno o varios parámetros. La tabla que presentamos a continuación presenta las distribuciones más comunes, los parámetros requeridos y sus valores por defecto. `n` representa siempre el tamaño de la muestra.

Muestras Aleatorias

Asociadas a estas funciones para generación de números aleatorios hay otras tres (para cada distribución) que permiten obtener valores de la densidad, la función de distribución y la función de cuantiles. En cada caso hay que reemplazar la letra inicial `r` por `d`, `p` o `q`, respectivamente. El argumento principal para estas funciones es el vector de puntos donde deseamos evaluarlas.

Muestras Aleatorias

Distribución	Función
Gaussiana	`rnorm(n, mean=0, sd=1)`
Exponencial	`rexp(n, rate=1)`
Gamma	`rgamma(n, shape, scale=1)`
Poisson	`rpois(n, lambda)`
Weibull	`rweibull(n, shape, scale=1)`
Cauchy	`rcauchy(n, location=0, scale=1)`
Beta	`rbeta(n, shape1, shape2)`
t	`rt(n, df)`
Fisher	`rf(n, df1, df2)`
χ^2	`rchisq(n, df)`
Binomial	`rbinom(n, size, prob)`
Multinomial	`rmultinom(n, size, prob)`
Geométrica	`rgeom(n, prob)`
Hipergeométrica	`rhyper(nn, m, n, k)`
Logística	`rlogis(n, location=0, scale=1)`

Muestras Aleatorias

Veamos algunos ejemplos.

```
> rnorm(10)
[1] -1.4984416 -1.0271529 -0.4484530
[4] -0.7705817 1.3690940 -2.4194197
[7] 1.1522398 -0.5763845 0.6463922 0.2162777
> rbinom(5, 20, 0.5)
[1] 11 9 12 9 10
> rexp(8)
[1] 0.86212931 0.07560819 0.07010501
[4] 5.11270242 1.02330924 1.51076381
[7] 1.54622939 0.24291849
> rpois(4, lambda=10)
[1] 10 12 10 7
```

Muestras Aleatorias

Buscamos ahora los cuantiles de la distribución normal típica para una prueba de dos colas al 5 %:

```
> qnorm(c(0.025,0.975))
[1] -1.959964 1.959964
```

Finalmente hacemos gráficos de la densidad y la función de distribución normal típica entre -3 y 3.

```
> puntos.x <- seq(-3,3,length=100)
> puntos.den <- dnorm(puntos.x)
> puntos.fd <- pnorm(puntos.x)
> plot(puntos.x,puntos.fd,type='l', xlab =
    'Valores', ylab='Densidad y Funcion de
    Distribucion', main='Dist. Normal')
> lines(puntos.x,puntos.den)
```

Ejercicio

Ejercicio 3

1. Genere un vector *x* con 10 números aleatorios de la distribución χ^2 con 5 grados de libertad.
2. Genere un vector *y* con 10 valores de la distribución de Poisson con parámetro 10.
3. Obtenga los cuantiles 95 y 99 de las distribuciones exponencial de parámetro 1, Cauchy con los parámetros por defecto, *t* con 10 grados de libertad y χ^2 con 20 grados de libertad.
4. Haga la gráfica de la densidad y la función de distribución *t* con dos grados de libertad entre -4 y 4.

Outline

Operaciones con vectores

R efectúa las operaciones aritméticas entre vectores componente a componente: si sumamos dos vectores de igual longitud el resultado es otro vector de la misma longitud, cuyas componentes son la suma de las componentes de los vectores que sumamos. De manera similar ocurre si realizamos cualquier otra operación aritmética, incluyendo la potenciación. Veamos algunos ejemplos.

```
> a <- 5:2
> b <- (1:4)*2
> a
[1] 5 4 3 2
> b
[1] 2 4 6 8
```

Operaciones con vectores

```
> a + b
[1] 7 8 9 10
> a - b
[1] 3 0 -3 -6
> a * b
[1] 10 16 18 16
> a / b
[1] 2.50 1.00 0.50 0.25
> a ^ b
[1] 25 256 729 256
```

Operaciones con vectores

Es posible realizar operaciones aritméticas entre un vector y un escalar:

```
> 2 * a
[1] 10  8  6  4
```

También es posible realizar operaciones con más de dos vectores simultáneamente.

```
> d <- rep(2,4)
> a + b + d
[1]  9 10 11 12
```

Operaciones con vectores

Veamos ahora un ejemplo más interesante. Supongamos que queremos evaluar la función

$$f(x, y) = \log\left(\frac{x^2 + 2y}{(x + y)^2}\right)$$

para varios valores de x e y. Una posibilidad es tomar cada par de valores y calcular $f(x, y)$. Podemos, sin embargo, aprovechar la forma en la cual trabaja `R` con vectores para realizar todas las operaciones con una sola evaluación. Comenzamos por crear los vectores que contienen los valores de interés para x e y.

Operaciones con vectores

```
> x <- 10:6
> y <- seq(1,9,2)
> x
[1] 10  9  8  7  6
> y
[1] 1 3 5 7 9
```

Definimos ahora la función en términos de estos vectores que hemos creado. Los resultados los guardaremos en un nuevo vector z.

```
> (z <- log( (x^2 + 2*y) / (x + y)^2 ))
[1] -0.1708177 -0.5039052 -0.8258336
[4] -1.1349799 -1.4271164
```

Operaciones con vectores

Funciones de uso común para vectores:

Nombre	**Operación**
`length()`	longitud
`sum()`	suma de las componentes del vector
`prod`	producto de las componentes del vector
`cumsum(), cumprod()`	suma y producto acumulados
`max(), min()`	máximo y mínimo del vector
`cummax(), cummin()`	máximo y mínimo acumulados
`sort()`	ordena el vector
`diff()`	calcula la diferencia entre las componentes
`which(x == a)`	vector de los índices de `x` para los cuales la comparación es cierta

Tabla 1.5. Funciones vectoriales.

Operaciones con vectores

Nombre	**Operación**
`which.max(x)`	índice del mayor elemento
`which.min(x)`	índice del menor elemento
`range(x)`	valores del mínimo y el máximo de `x`
`mean(x)`	promedio de los elementos de `x`
`median(x)`	mediana de los elementos de `x`
`round(x,n)`	redondea los elementos de `x` a `n` decimales
`rank(x)`	rango de los elementos de `x`
`unique(x)`	vector con las componentes de `x` sin repeticiones

Tabla 1.5. Funciones vectoriales.

Ejercicio

Ejercicio 4

1. Genere un vector de nombre `np` cuyas componentes sean 100 números generados al azar de la distribución de Poisson de parámetro 10.
2. Obtenga el máximo, mínimo, media, mediana y rango de `np`.
3. ¿Cuántos valores distintos hay en el vector?
4. ¿En qué lugar se encuentran el máximo y el mínimo?
5. Calcule los valores de la siguiente función f en los pares (x_i, y_i) con componentes tomadas a partir de los vectores que obtuvo en las preguntas 1 y 2 del ejercicio 3.

$$f(x, y) = \exp\{x^y - \log(x * y)\}$$

Outline

Atributos

Toda expresión que escribimos en la consola es interpretada por R y produce un valor. Este valor es un objeto de datos ('data object') al cual puede asignársele un nombre. Los objetos de datos están compuestos por elementos, que en objetos simples corresponden a datos individuales y en objetos más complejos pueden consistir de otros objetos de datos.

Atributos

Los elementos pueden ser:

- Lógicos: Toman los valores `T` (cierto) o `F` (falso).
- Numéricos: números de punto flotante (con precisión doble). Los valores numéricos pueden escribirse como enteros (por ejemplo `4, -24`), decimales (`3.14, -2.717`) o en notación científica (`4e-23`).
- Complejos: números complejos de la forma `a + bi`, donde `a` y `b` son numéricos (por ejemplo `3.6 + 2.4i`).
- Caracteres: Sucesiones de caracteres limitados por comillas simples (`' '`) o dobles (`" "`) (por ejemplo `'letra' "prueba"`).

Atributos

Esta característica que hemos descrito se conoce como el modo (’mode’) de los elementos. Hemos listado los modos progresando del menos informativo al más informativo. Este orden es importante al considerar objetos de datos creados a partir de elementos de distintos modos, ya que hay objetos de datos que no permiten combinar elementos con modos distintos.

Atributos

El número de elementos de un objeto determina su longitud ('length'). Los objetos de datos más simples, los vectores, se clasifican según su modo y longitud. Se puede crear un vector de longitud 1 escribiendo el elemento y presionando 'enter'

```
35.7
[1] 35.7
> 'hola'
[1] "hola"
```

Atributos

Para combinar varios elementos en un vector usamos la función `c` o la función `scan`

```
> c(T,T,T,F)
[1] TRUE TRUE TRUE FALSE
> c(21.5,55.3,12)
[1] 21.5 55.3 12.0
> scan()
1: 2.3 55.5 567
4:
Read 3 items
[1] 2.3 55.5 567.0
```

Atributos

Si se combinan elementos de modos distintos en un mismo vector, R le asigna a todos los elementos el modo más informativo:

```
> c(T,F,12.3)
[1] 1.0 0.0 12.3
> c(8.3,12+3i)
[1] 8.3+0i 12.0+3i
> c(F,12.3,"hola")
[1] "FALSE" "12.3" "hola"
```

Atributos

Para conocer el modo y longitud de cualquier objeto de datos podemos usar las funciones `mode` y `length` :

```
> mode(c(F,12.3,"hola"))
[1] "character"
> length(c(T,T,T,F))
[1] 4
```

Atributos

El modo y la longitud son *atributos* del objeto. Estos dos atributos están presentes en todos los objetos de datos y se llaman atributos implícitos. Los vectores sólo tienen estos dos atributos. Otros atributos especifican la estructura y los aspectos distintivos de otros tipos de objetos. Por ejemplo, si introducimos una estructura multidimensional a los elementos de un vector añadiendo el atributo '`dim`', creamos un objeto llamado arreglo ('array'). El atributo `dim` es un vector numérico que especifica cuantos elementos deben colocarse en cada dimensión. La longitud del atributo `dim` determina cuántas dimensiones hay. Si `dim` tiene longitud 2 el arreglo se llama matriz ('matrix'). En este caso el primer elemento determina el número de filas y el segundo, el número de columnas.

Atributos

```
> x <- 1:10
> x
 [1]  1  2  3  4  5  6  7  8  9 10
> dim(x) <- c(2,5)
> x
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    3    5    7    9
[2,]    2    4    6    8   10
```

Atributos

Para cualquier modo, los datos faltantes se indican por `NA` ('not available'). Es posible usar la notación exponencial para valores numéricos grandes

```
> N <- 6.02e23
> N
[1] 6.02e+23
```

Atributos

`R` puede manejar correctamente valores infinitos ($\pm\infty$) con `Inf` y `-Inf` o valores no numéricos con `NaN` ('not a number').

```
> x <- 2/0
> x
[1] Inf
> exp(x)
[1] Inf
> exp(-x)
[1] 0
> x - x
[1] NaN
```

Atributos

Objeto	Modos	Varios modos
Vector	Numérico, carácter, complejo o lógico	No
Factor	Numérico o carácter (categórico)	No
Matriz	Numérico, carácter, complejo o lógico	No
Arreglo	Numérico, carácter, complejo o lógico	No
Cuadro de datos	Numérico, carácter, complejo o lógico	Si
ts	Numérico, carácter, complejo o lógico	No
Lista	Numérico, carácter, complejo, lógico, función, ...	No

Tabla 1.6. Tipos de objetos.

Atributos

Un factor es una variable categórica. Una matriz es una tabla con dos dimensiones. Un arreglo es una tabla con *k* dimensiones. Para una matriz o un arreglo los elementos deben ser todos del mismo tipo. Un cuadro de datos ('data frame') es una tabla formada por vectores y/o factores de la misma longitud pero posiblemente de modos distintos. Un 'ts' es un conjunto de datos correspondiente a una serie temporal y tiene atributos adicionales como frecuencia y fechas. Finalmente, uno de los objetos más útiles es la lista ('`list`'), que puede ser usada para combinar cualquier colección de objetos de datos en un solo objeto (incluyendo otras listas).

Outline

Matrices

Una matriz es un arreglo rectangular de celdas, cada una de las cuales contiene un valor. Para crear una matriz en `R` es posible usar la función `matrix`, cuya sintaxis es `matrix(datos, nrow, ncol, byrow=F, dimnames = NULL)`, donde `nrow` y `ncol` representan, respectivamente, el número de filas y columnas de la matriz. Sólo el primer argumento es indispensable. Si no aparecen ni el segundo ni el tercero, los datos se colocan en una matriz unidimensional, es decir, en un vector. Si sólo uno de los valores se incluye, el otro se determina por división y se asigna.

Matrices

```
> matrix(1:6)
     [,1]
[1,]    1
[2,]    2
[3,]    3
[4,]    4
[5,]    5
[6,]    6
> matrix(1:6, nrow=3)
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
```

Matrices

Veamos como se construyó esta última matriz. Sus elementos son los números del 1 al 6, listados en ese orden, y queremos formar una matriz de tres filas. `R` hace la división del número de elementos entre el número de filas y obtiene que el número de columnas es 2. Para `R` los vectores son vectores columna y por esta razón los elementos de la lista se introducen primero en la primera columna y luego en la segunda. Es importante tener este orden en cuenta a la hora de hacer un programa.
Si quisiéramos llenar la matriz por filas, es necesario poner `'T'` como valor de `byrow`:

```
matrix(1:6, nrow=3, byrow=T)
     [,1] [,2]
[1,]    1    2
[2,]    3    4
[3,]    5    6
```

Matrices

Los elementos de una matriz deben ser todos del mismo tipo. Para guardar columnas de distintos tipos en un arreglo bidimensional es necesario usar un cuadro de datos (data frame) que estudiaremos un poco más adelante.

Matrices

Para sumar y restar matrices, operaciones que se realizan componente a componente, la única condición necesaria es que ambas matrices tengan las mismas dimensiones. Los símbolos usuales de suma y resta se usan para estas operaciones.

```
> (A <- matrix(1:6, nrow=3, byrow=T))
     [,1] [,2]
[1,]    1    2
[2,]    3    4
[3,]    5    6
> (B <- matrix(seq(0, 10, 2), 3, 2))
     [,1] [,2]
[1,]    0    6
[2,]    2    8
[3,]    4   10
```

Matrices

```
> A + B
     [,1] [,2]
[1,]    1    8
[2,]    5   12
[3,]    9   16
> A - B
     [,1] [,2]
[1,]    1   -4
[2,]    1   -4
[3,]    1   -4
```

Matrices

A pesar de la regla sobre la coincidencia de las dimensiones de las matrices que van a ser sumadas o restadas, es posible sumar o restar un escalar a una matriz:

```
> A + 2
     [,1] [,2]
[1,]    3    4
[2,]    5    6
[3,]    7    8
```

También es posible multiplicar o dividir por un escalar:

```
> B / 2
     [,1] [,2]
[1,]    0    3
[2,]    1    4
[3,]    2    5
```

Matrices

La multiplicación de matrices es una operación más complicada y no se efectúa componente a componente. La sintaxis para la multiplicación de matrices es la siguiente:

```
A %*% B
```

La regla para la multiplicación de matrices es que el número de columnas de la primera matriz debe ser igual al número de filas de la segunda. Por ejemplo, la matriz A que hemos definido anteriormente es una matriz 3 x 2. La podemos multiplicar por cualquier matriz que tenga 2 filas:

Matrices

```
> D <- matrix ((1:8)*3,2)
> D
     [,1] [,2] [,3] [,4]
[1,]    3    9   15   21
[2,]    6   12   18   24
> A %*% D
     [,1] [,2] [,3] [,4]
[1,]   15   33   51   69
[2,]   33   75  117  159
[3,]   51  117  183  249
```

Matrices

También es posible multiplicar dos matrices componente a componente, siempre que tengan iguales dimensiones. Para esto usamos el símbolo usual * de la multiplicación:

```
> A * B
     [,1] [,2]
[1,]    0   12
[2,]    6   32
[3,]   20   60
```

Matrices

La siguiente tabla presenta otras operaciones comunes con matrices

Nombre	**Operación**
`dim()`	dimensiones de la matriz
`as.matrix()`	obliga al argumento a que opere como una matriz
`t()`	trasposición de matrices
`diag()`	extrae los elementos de la diagonal
`det()`	determinante
`solve()`	inversa de una matriz
`eigen()`	calcula eigenvalores y eigenvectores

Tabla 1.7. Funciones matriciales.

Matrices

Veamos como usar algunas de estas funciones. Primero obtenemos la traspuesta de una matriz y operamos con ella.

```
> XX <- matrix(c(2,3,4,1,5,3),ncol=3)
> XX
     [,1] [,2] [,3]
[1,]    2    4    5
[2,]    3    1    3

> t(XX)
     [,1] [,2]
[1,]    2    3
[2,]    4    1
[3,]    5    3
```

Matrices

```
> XX %*% t(XX)
     [,1] [,2]
[1,]   45   25
[2,]   25   19
> t(XX) %*% XX
     [,1] [,2] [,3]
[1,]   13   11   19
[2,]   11   17   23
[3,]   19   23   34
```

Matrices

Para hallar las columnas o filas de una matriz podemos usar las funciones `col` y `row`

```
> col(XX)
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    1    2    3
> row(XX)
     [,1] [,2] [,3]
[1,]    1    1    1
[2,]    2    2    2
```

Matrices

La función `diag` extrae los elementos de la diagonal de una matriz,

```
> diag(XX)
[1] 2 1
> diag(t(XX) %*% XX)
[1] 13 17 34
```

y también sirve para crear matrices diagonales:

```
> diag(1:4)
     [,1] [,2] [,3] [,4]
[1,]    1    0    0    0
[2,]    0    2    0    0
[3,]    0    0    3    0
[4,]    0    0    0    4
> diag(3)
     [,1] [,2] [,3]
[1,]    1    0    0
[2,]    0    1    0
[3,]    0    0    1
```

Matrices

Ahora creamos una matriz cuadrada para ver el uso de la función `solve`

```
> YY <- matrix(c(12,3,8,16,21,5,7,9,12,18,
      4,3,19,5,21,8) ,ncol=4)
> YY
     [,1] [,2] [,3] [,4]
[1,]   12   21   12   19
[2,]    3    5   18    5
[3,]    8    7    4   21
[4,]   16    9    3    8
```

Matrices

Para hallar la inversa de `YY` escribimos

```
> solve(YY)
             [,1]         [,2]          [,3]         [,4]
[1,] -0.04257496  0.012984969  0.0004721807  9.176045e-02
[2,]  0.09056032 -0.041945384 -0.0606161958 -2.974738e-02
[3,] -0.01497206  0.066026600 -0.0021444873 -7.869678e-05
[4,] -0.01111592 -0.003541355  0.0680530416 -2.502558e-02
```

y verificamos multiplicando las matrices,

```
> YY %*% solve(YY)
              [,1]          [,2]          [,3]          [,4]
[1,]  1.000000e+00 -1.040834e-17 -8.326673e-17 -9.367507e-17
[2,]  0.000000e+00  1.000000e+00  2.775558e-17 -1.734723e-17
[3,]  4.163336e-17 -1.734723e-17  1.000000e+00 -1.006140e-16
[4,] -1.387779e-17  1.387779e-17  0.000000e+00  1.000000e+00
```

Matrices

```
> round(YY %*% solve(YY),10)
     [,1] [,2] [,3] [,4]
[1,]    1    0    0    0
[2,]    0    1    0    0
[3,]    0    0    1    0
[4,]    0    0    0    1
> round(solve(YY) %*% YY,10)
     [,1] [,2] [,3] [,4]
[1,]    1    0    0    0
[2,]    0    1    0    0
[3,]    0    0    1    0
[4,]    0    0    0    1
```

Matrices

La misma instrucción `solve` usada con una matriz `A` y un vector columna `b` permite resolver la ecuación lineal `b=Ax` para obtener `x`

```
> b <- c(1,2,3,4)
> (x <- solve(YY,b))
[1] 0.35185331 -0.29416857 0.11033289
0.08585819
> YY %*% x
     [,1]
[1,]    1
[2,]    2
[3,]    3
[4,]    4
```

Ejercicio

Ejercicio 5

1. Defina una matriz de nombre `AA` de tamaño 4×4 con números seleccionados por usted.
2. Halle la matriz inversa con dos decimales de precisión y verifique que el producto de las matrices en ambos órdenes da la identidad.
3. Resuelva el sistema de ecuaciones $AA\mathbf{x} = \mathbf{b}$, para $\mathbf{b} = (1, 2, 4, 3)$.
4. Obtenga los eigenvectores y eigenvalores de `AA`.

Matrices

Veamos un ejemplo un poco más complicado para fijar las ideas.

Compañía	Producción	Reservas	Refin.	Ventas
Exxon Mobil	4,3	21500	6,6	8,9
Shell	3,7	20500	3,4	5,7
BP Amoco	4,1	19400	3,3	5,4
Total ELF	2,1	9600	2,4	3,2

Tabla 1.8

Supongamos que tenemos datos sobre la producción, reservas, capacidad de refinación y ventas de productos refinados, todos medidos en millones de barriles por día, de cuatro compañías petroleras internacionales y queremos almacenar estos datos en una matriz.

Matrices

```
> datos <- c(4.3, 21500, 6.6, 8.9, 3.7,
    20500, 3.4, 5.7, 4.1, 19400, 3.3,
    5.4, 2.1, 9600, 2.4, 3.2)
> petro.datos <- matrix(datos, nrow=4,
    byrow = T)
> petro.datos
     [,1]  [,2] [,3] [,4]
[1,]  4.3 21500  6.6  8.9
[2,]  3.7 20500  3.4  5.7
[3,]  4.1 19400  3.3  5.4
[4,]  2.1  9600  2.4  3.2
```

Matrices

Esta matriz que hemos construido contiene la información que hemos descrito, pero no es muy útil porque no sabemos que variable representa cada columna ni que compañía corresponde a cada fila. Para resolver esto podemos guardar los nombres correspondientes usando una función llamada `dimnames`. Esto permite recordar qué hemos almacenado en cada lugar y obtener una mejor representación de los datos. Además, los nombres asignados pueden ser usados para tener acceso a los datos.

Matrices

Si la matriz tiene n columnas y p filas, los dos vectores que se usan para los nombres deben tener esas mismas dimensiones. Cada componente de estos vectores es una sucesión de caracteres que describe el contenido correspondiente. Si antes de asignar nombres pedimos el contenido de `dimnames` obtenemos

```
> dimnames()
NULL
```

`NULL` significa vacío.

Matrices

Veamos ahora la dimensión de la matriz `petro.datos`

```
dim(petro.datos)
[1] 4 4
```

Asignamos ahora los nombres de las compañías y de las variables a dos vectores y los copiamos usando la función `dimnames`

```
> nombres <- c("Exxon Mobil", "Shell",
    "BP Amoco", "Total ELF")
> variables <- c("Prod.","Reservas","Refin.",
    "Ventas")
> dimnames (petro.datos) <- list(nombres,
    variables)
```

Matrices

```
> petro.datos
             Prod. Reservas Refin. Ventas
Exxon Mobil   4.3    21500    6.6    8.9
Shell         3.7    20500    3.4    5.7
BP Amoco      4.1    19400    3.3    5.4
Total ELF     2.1     9600    2.4    3.2
```

Matrices

Con esta opción, las filas y columnas tienen nombres que permiten identificar claramente los datos. Es posible, también, ponerle sólo nombres a las filas o a las columnas, sin incluir los otros:

```
> dimnames(petro.datos) <- list(nombres,
    NULL)
> petro.datos
             [,1]   [,2]  [,3]  [,4]
Exxon Mobil 4.3   21500   6.6   8.9
Shell        3.7   20500   3.4   5.7
BP Amoco     4.1   19400   3.3   5.4
Total ELF    2.1    9600   2.4   3.2
```

Matrices

```
> dimnames(petro.datos) <- list(NULL, variables)
> petro.datos
     Prod. Reservas Refin. Ventas
[1,]   4.3     21500   6.6      8.9
[2,]   3.7     20500   3.4      5.7
[3,]   4.1     19400   3.3      5.4
[4,]   2.1      9600   2.4      3.2
> dimnames(petro.datos) <- list(nombres,
    variables)
> dimnames(petro.datos)
[[1]]: [1] "Exxon Mobil" "Shell" "BP Amoco"
    "Total ELF"
[[2]]: [1] "Prod." "Reservas" "Refin." "Ventas"
```

Matrices

Para tener acceso a los datos es necesario conocer la fila y columna en la que se encuentran. Por ejemplo, sabemos que Shell está en la segunda fila y que la ventas están en la tercera columna. Para hallar las ventas de Shell buscamos el elemento [2,4] de la matriz

```
> petro.datos[2,4]
[1] 5.7
```

y obtenemos que las ventas de Shell son de 5.7 millones de barriles diarios. Observamos que hay que especificar primero el número de la fila y luego el de la columna. Para ver todos los elementos de una fila hay que poner el número de la fila y todos los números de las columnas. Algo similar hay que hacer para las columnas.

Matrices

Para ver los datos de BP Amoco,

```
> petro.datos [3, 1:4]
Prod. Reservas Refin. Ventas
 4.1      19400   3.3      5.4
```

Una alternativa económica a esta notación es dejar en blanco las columnas (o filas),

```
> petro.datos [3, ]
Prod. Reservas Refin. Ventas
 4.1      19400   3.3      5.4
```

También es posible usar los `dimnames` en lugar de los números de filas y columnas, usando comillas para los nombres:

```
> petro.datos ["Total ELF", "Reservas"]
[1] 9600
> petro.datos [ ,"Prod."]
Exxon Mobil Shell BP Amoco Total ELF
        4.3   3.7      4.1       2.1
```

Matrices

De manera similar es posible cambiar una componente de la matriz. Por ejemplo, si queremos cambiar la producción de Shell de 3.7 a 3.9 millones de barriles diarios podemos hacerlo de la siguiente manera:

```
> petro.datos["Shell", "Prod."]
[1] 3.7
> petro.datos["Shell", "Prod."] <- 3.9
> petro.datos["Shell", "Prod."]
[1] 3.9
> petro.datos["Shell", "Prod."] <- 3.7
> petro.datos["Shell", "Prod."]
[1] 3.7
```

Al final, hemos regresado al valor original.

Unión de Matrices

Supongamos que queremos añadir información sobre otra compañía petrolera, Chevron, a la matriz `petro.datos` que construimos anteriormente. Podemos crear una matriz con los datos faltantes y 'unirla' a la matriz existente. Para añadir filas nuevas se usa la función `rbind` mientras que para añadir columnas se usa `cbind` (la inicial r es por 'row' mientras que c es por 'column').

```
> Chevron <- c(1.5, 6200, 1.6, 2.0)
> Chevron
[1] 1.5 6200.0 1.6 2.0
```

Matrices

```
> petro.datos <- rbind(petro.datos, Chevron)
> petro.datos
             Prod. Reservas Refin. Ventas
Exxon Mobil   4.3     21500    6.6      8.9
Shell         3.7     20500    3.4      5.7
BP Amoco      4.1     19400    3.3      5.4
Total ELF     2.1      9600    2.4      3.2
Chevron       1.5      6200    1.6      2.0
```

Observamos que `R` toma como `dimname` de la nueva fila el nombre del archivo que se unió a la matriz. Para añadir una columna se usa exactamente el mismo procedimiento con la función `cbind`. También es posible añadir múltiples filas o columnas pero hay que prestar atención las dimensiones para evitar problemas.

Problema

Ejercicio 5

1. Construya una matriz con la siguiente información

	GDP	Pob	Inflacion
Austria	208	8	2.4
Francia	1432	61	1.7
Alemania	2112	82	2.0
Suiza	259	7	1.6

2. Obtenga los valores para Francia.
3. Obtenga una lista de la población de todos estos países.