

Curso básico de



Agosto de 2016

¿Qué es *R*?

R es un conjunto integrado de programas para manipulación de datos, cálculos y gráficos. Se trata de un sistema para análisis estadísticos creado por Ross Ihaka y Robert Gentleman. *R* tiene una naturaleza doble: es un programa y un lenguaje de programación, desarrollado originalmente como una versión libre del lenguaje *S*, creado por los Laboratorios AT&T Bell.

Entre otras características *R* dispone de:

- Almacenamiento y manipulación efectiva de datos, operadores para cálculo sobre variables indexadas (Arrays), en particular matrices,



R dispone de ...

- Una amplia, coherente e integrada colección de herramientas para análisis de datos,
- Enormes posibilidades gráficas para análisis de datos, que funcionan directamente sobre pantalla o impresora, y
- Un lenguaje de programación bien desarrollado, simple y efectivo, que incluye condicionales, ciclos, funciones recursivas y posibilidad de entradas y salidas.



Además...

- R se distribuye gratuitamente bajo los términos de la *GNU General Public Licence*; su desarrollo y distribución son llevados a cabo por varios estadísticos conocidos como el *Grupo Nuclear de Desarrollo de R*.
- R está disponible en varias formas: el código fuente escrito principalmente en C (y algunas rutinas en Fortran), esencialmente para máquinas Unix y Linux, o como archivos precompilados para Windows, Linux, Macintosh y Alpha Unix.
- Los archivos necesarios para instalar R, ya sea desde las fuentes o archivos precompilados, se distribuyen desde el sitio de internet *Comprehensive R Archive Network (CRAN)*, junto con las instrucciones de instalación.

<https://cran.r-project.org/>

¿Cómo funciona R?

- Lenguaje Orientado a Objetos: ***simplicidad y flexibilidad***, por dos razones: es interpretado (como Java) y no compilado (como C, C++, Fortran), lo cual significa que los comandos escritos en el teclado son ejecutados directamente sin necesidad de construir ejecutables.
- La sintaxis de *R* es muy simple e intuitiva.



La consola de R

- Al abrir el programa se abre una ventana llamada “R Console”, que llamaremos la consola. En esta ventana se escriben las instrucciones que ejecutará el programa.
- El símbolo “>” en la consola es el apuntador (prompt) e indica que el programa está listo para recibir instrucciones.
- En lo que sigue escribiremos los comandos de *R* precedidos por el símbolo del apuntador, y usaremos letras de tipo Courier New para destacar el texto que se escribe en la consola.
- La consola es útil para comandos sencillos que van a ser ejecutados de inmediato.



El directorio de trabajo

Por ahora vamos a trabajar desde la consola

- Para comenzar vamos a considerar el directorio de trabajo.
- Cuando estamos llevando varios proyectos es conveniente tener directorios de trabajo separados, uno para cada proyecto, para que no haya confusiones con nombres de variables, funciones, conjuntos de datos, etc.
- Para averiguar en cual directorio estamos escribimos

```
> getwd()
```

El directorio de trabajo

- Una manera de cambiar el directorio es usando la instrucción `setwd()`, aunque es más sencillo usar la opción del menú: Archivo > Cambiar dir
- También es posible ir al menú *Archivo* y utilizar las opciones que aparecen.
- 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.

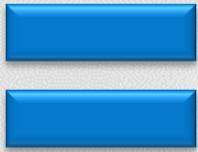
Librerías

- *R* es una plataforma de programación que constantemente se encuentra actualizándose. Se trata de un software libre, pero no solamente para acceder al mismo, sino también para hacer contribuciones a mejorarlo. Para ello, *R* cuenta con un repositorio en su pagina web donde usuarios especializados de todo el mundo pueden compartir *librerías* (*packages*) desarrolladas por ellos mismos, y que son paquetes accesorios para *R* a manera de programas adicionales a la plataforma base, cuya finalidad es realizar funciones específicas en *R*.

Analogía



Analogía



!!

Ya en serio...

- Para instalar estas librerías hacemos lo siguiente:
 - 1) En el menú de *R*, seleccionar *Paquetes*
 - 2) Dar clic en *Instalar paquetes* (se necesita conexión de internet)
 - 3) Se abre una ventana donde debe seleccionarse un sitio web espejo de *R* (*mirror*), que es un servidor que almacena bases de datos relacionadas con *R* (librerías entre otras cosas). Puede seleccionarse cualquiera de ellos.
 - 4) Una vez elegido el espejo aparece una lista donde se selecciona la librería a instalar. Una vez seleccionada, *R* instala la librería.
 - 5) También pueden instalarse librerías en forma local (sin internet). Para ello es necesario contar con una carpeta comprimida que contenga la librería deseada, en alguna ubicación dentro del equipo. Para hacerlo se da clic en *Instalar paquete(s) a partir de archivos locales zip...*

Ejercicio 1

- Instala la librería *foreign* de la carpeta comprimida que se encuentra en la carpeta del curso, llamada *foreign_0.8-66.tar.gz*.
- Para hacerlo simplemente selecciona el menú *Paquetes > Instalar paquete(s) a partir de archivos locales zip...*
- Busca la ubicación del paquete, selecciónalo y listo.
- Invoca la librería tecleando la instrucción *library(foreign)*

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 simplemente desplegar dicho conjunto de datos:

```
> cars
```

	speed	dist
--	-------	------

1	4	2
---	---	---

2	4	10 ...
---	---	--------

Expresiones comunes

- 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 ahora 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
```

Operadores

- 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.

Jerarquía de operaciones

- 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
```

Funciones útiles

- Algunas funciones de uso común disponibles en *R* son las siguientes:

Nombre	Operación
<code>abs()</code>	valor absoluto
<code>sin()</code> , <code>cos()</code> , <code>tan()</code>	funciones trigonométricas (en rads.)
<code>pi</code>	el número $\pi = 3,1415926\dots$
<code>exp()</code> , <code>log()</code>	exponencial y logaritmo
<code>factorial()</code>	la función factorial
<code>choose()</code>	número combinatorio.
<code>round()</code>	redondea el número de decimales
<code>ceiling()</code>	calcula el menor entero mayor o igual que el módulo
<code>floor()</code>	calcula el mayor entero menor o igual que el módulo
<code>% %</code>	
<code>% / %</code>	división entera

Tabla 1. Funciones numéricas.

Ejercicio 2

- 1 Establece la carpeta del curso como carpeta de trabajo.
- 2 Calcula las siguientes expresiones en *R*:
 1. El valor absoluto de -12
 2. La raíz cuadrada del número pi
 3. El valor de 4 elevado a la potencia 5
 4. $8439-3798$ por $2764+4664$
 5. El módulo de 29 entre 5
 6. La división entera de 94 entre 9
 7. El factorial de 6

Índice en línea de comandos

- Observamos que en los ejemplos anteriores la respuesta incluye el índice, que es un número entero escrito entre corchetes que aparece siempre al inicio del renglón.
- 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.

```
> 87:129
```

```
[1]  87  88  89  90  91  92  93  94  95  96  97  98  
[13] 99 100 101 102 103 104 105 106 107 108 109 110  
[25] 111 112 113 114 115 116 117 118 119 120 121 122  
[37] 123 124 125 126 127 128 129
```

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

```
> sqrt(16) # El instructor es muy guapo  
[1] 4
```

Flechas de desplazamiento

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.

Corregir una línea

- Como ejemplo, supongamos que queremos calcular una expresión con una raíz cuadrada, pero cometemos un error al escribir `sqr`:

```
> 94**2-sqr(433)
```

Error: no se pudo encontrar la función "sqr"

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

```
> 94**2-sqr(433)
```

```
[1] 8815.191
```

Asignación

- Uno de los operadores de uso más frecuente es el de asignación, que asocia valores con objetos, 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
```

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

```
> b = -5
```

Asignación

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

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

- Cuando hacemos una asignación, *R* no muestra 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
```

Remover objetos

- Todas las asignaciones permanecen en la memoria RAM 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: objeto 'a' no encontrado
```

- *R* guarda automáticamente las asignaciones de valores a variables, de modo que las variables pasan a ser objetos y permanecen en el directorio de trabajo a menos que sean eliminadas. Para borrar todos los objetos en la memoria escribimos

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

Reasignación de valores

- Al reasignar un valor a un objeto, se borra el valor anterior. Esto es particularmente útil para la programación de ciclos y funciones, pero se debe tener cuidado para no efectuar reasignaciones erróneas.

```
> a <- 3  
> a  
[1] 3  
> a <- 7  
> a  
[1] 7
```

Reto:

Asigna un valor numérico dado a un objeto “a” y otro a un objeto “b”. Luego, utilizando reasignaciones en combinación con alguna operación matemática básica, intercambia los valores de “a” y “b”, es decir, al final “b” debe quedar con el valor que tenía al inicio el objeto “a”, y “a” debe quedar con el valor que tenía “b”. Nota: NO debes crear objetos adicionales (por ejemplo, un objeto “c”, como apoyo para las reasignaciones).

Pista: $a+b$

Guardado de objetos

- Al cerrar el programa aparece un mensaje preguntando si se desea guardar una 'imagen de área de trabajo'. Si se hace 'click' en 'Sí' 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 un 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("nombre_de_archivo.RData")`.

Ayuda de R

- 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(median)`abre una ventana que describe con detalle como se calcula la mediana y presenta referencias y ejemplos.
- Otra manera de escribir esta instrucción es
 - > `? median`

Ayuda de R

- Una manera alternativa de obtener ayuda es ejecutar el comando
> `help.start()`
- 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 búsqueda.

Ayuda de R

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

```
> example(image)
```

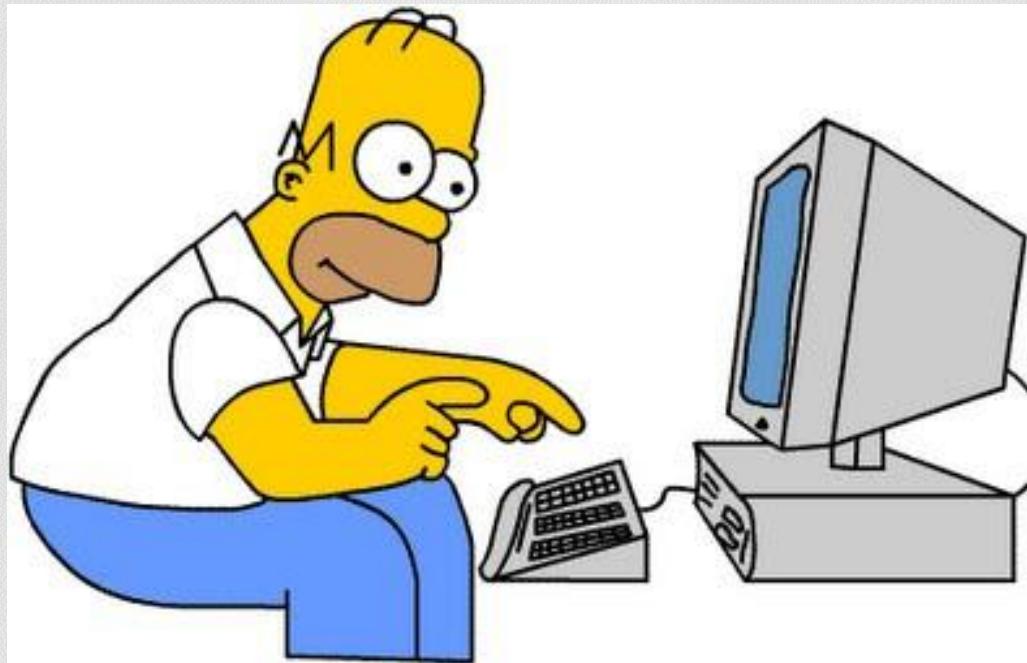
produce una serie de cuatro gráficas en una misma ventana. Para que aparezca la siguiente gráfica es necesario apretar la tecla Enter.

Ahora vamos a trabajar con un script en Rstudio

- La consola es útil para comandos sencillos que van a ser ejecutados de inmediato, pero con frecuencia resulta más conveniente usar las ventanas script, que permiten escribir instrucciones sin que se ejecuten de manera automática, y de esta manera, desarrollar un guión o 'script' constituido por una serie de instrucciones en secuencia.
- Estas ventanas pueden ser guardadas para ser ejecutadas o modificadas posteriormente. Para abrir una ventana de este tipo abrimos el menú 'Archivo' y seleccionamos 'Nuevo script'.
- Las instrucciones que se escriben sobre esta nueva ventana no se ejecutan automáticamente al presionar la tecla 'Enter', sino que se debe seleccionar la instrucción y luego presionar 'Control+R', o bien presionar el botón derecho del ratón y seleccionar 'Correr linea o seleccionar'.
- Las instrucciones seleccionadas se ejecutan en la consola y los resultados aparecen allí.

Scripts

- Lo que vamos a realizar en esta primera sesión se encuentra en un script llamado *guion.R*. Para abrirlo vamos al menú *Archivo* y seleccionamos *Abrir script*. Luego abrimos la Carpeta del curso *R* y en ella el archivo *guion.R*



Ejercicio 3

- Abre el script llamado guion.R
- Selecciona y corre todas las líneas de código correspondientes al Ejercicio 3.
- Observa los resultados en la consola y el gráfico que se genera.

Tipos y Estructura de Datos

- Toda expresión que escribimos en la consola es interpretada por *R* y produce un valor. Este valor es un objeto de datos 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.

Tipos de elementos

Los elementos pueden ser:

- Lógicos: Toman los valores T (cierto) o F (falso).
- Numéricos: 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").

Tipos de elementos

- Esta característica que hemos descrito se conoce como el *modo* o *tipo* ('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.

Tipos de elementos

- 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"
```

Vectores

- 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: 24
2: 55
3: 67
4:
Read 3 items
[1] 24 55 67
```

Tipo de datos en un vector

- 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"
```

Modo y longitud

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

```
> mode(c(F, 12.3, "holá"))
[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.

El atributo “dim”

- 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.

El atributo “dim”

- 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.

- ```
> a <- c(34, 8, 2, 54, 49, 1)
```

```
> a
```

```
[1] 34 8 2 54 49 1
```

```
> dim(a) <- c(2, 3)
```

```
> a
```

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

```
[1,] 34 2 49
```

```
[2,] 8 54 1
```

# Notación

---

- 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
```

# Datos indefinidos

---

- 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
```

# *Tipos de Objetos*

| <b>Objeto</b>   | <b>Modos</b>                                           | <b>Varios modos</b> |
|-----------------|--------------------------------------------------------|---------------------|
| 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                  |
| Serie de Tiempo | Numérico, carácter, complejo o lógico                  | No                  |
| Lista           | Numérico, carácter, complejo o lógico,<br>función, ... | Si                  |

**Tabla 2. Tipos de objetos.**

# Definiciones

---

- 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.
- Una *Serie de Tiempo* 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).

# *Identificar tipos de objetos*

---

- Podemos forzar e identificar un tipo de objeto:

```
> a <- c("alto", "medio", "bajo")
> a
[1] "alto" "medio" "bajo"
> is.vector(a)
[1] TRUE
> a <- as.factor(a)
> a
[1] alto medio bajo
Levels: alto bajo medio
> is.vector(a)
[1] FALSE
> is.factor(a)
[1] TRUE
```

# *Identificar tipos de objetos*

---

```
> a <- as.matrix(a)
> a
 [,1]
[1,] "alto"
[2,] "medio"
[3,] "bajo"
> is.factor(a)
[1] FALSE
> is.matrix(a)
[1] TRUE
```

# Lectura de datos

- Vamos a leer una tabla guardada en un archivo de texto usando la función `read.table()`. En la carpeta del curso hay un archivo llamado `MEXpob.txt` que tiene una tabla que muestra la población de México (en millones de habitantes), 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

---

- 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

```
> read.table(file.choose(), header = T)
```

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

---

- Una función de lectura de datos que *R* tiene por default es *read.table()*, que permite leer tablas a partir de archivos de texto (.txt), cuyos datos están separados por espacios o saltos de línea (en forma tabular), como el archivo `MEXPob.txt` que vimos en el ejemplo.
- Otra función para leer datos es *read.csv()*. Esta función permite leer archivos con datos separados por comas. La versatilidad del formato .csv radica en que minimiza los errores de lectura, y es manejable en Excel. De hecho cualquier tabla de Excel (.xls ó .xlsx) puede guardarse en formato .csv desde dicho programa, para leerse en *R*.
- La librería *foreign* que instalamos en el Ejercicio 1, permite leer datos en *R* desde varios otros formatos: SPSS (*read.spss*), DBF (*read.dbf*), Stata (*read.dta*), Minitab (*read.mtp*) y SAS (*read.ssd*), entre otros.

# Lectura de datos

---

```
> read.dbf('MEXPob.dbf')
 ANIO POB
1 1895 12.6
2 1900 13.6
3 1910 15.2
4 1921 14.3
5 1930 16.6
6 1940 19.7
7 1950 25.4
8 1960 34.9
9 1970 48.2
10 1980 66.8
11 1990 81.2
12 1995 91.2
13 2000 97.4
```

# *Creación de tablas de datos*

---

- Los datos de la Tabla 3 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.

| <b>Estiramiento</b> | <b>Distancia</b> |
|---------------------|------------------|
| 46                  | 148              |
| 54                  | 182              |
| 48                  | 173              |
| 50                  | 166              |
| 44                  | 109              |
| 42                  | 141              |
| 52                  | 166              |

**Tabla 3: Distancia (cm) recorrida por una banda elástica en función del estiramiento (cm).**

# Creación de tablas de datos

---

- 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))
```

# *Creación de tablas de datos*

---

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

# *Edición de tablas de datos*

---

- Es posible usar una 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.

## Ejercicio 4

- 1 Lee el archivo de datos *equipaje.txt* que se encuentra en el directorio de trabajo y guárdalo en un objeto con el nombre *equipaje*.
- 2 Analiza su estructura usando los comandos `str()`, `mode()`, `dim()`, `length()`, `is.list()`, `is.data.frame()`.
- 3 Ábrelo con el editor de textos y cambie el dato 9.5 por 9.3.
- 4 Añade las siguientes componentes al final del archivo: 7.8, 6.9, 8.2
- 5 Cierra el archivo.

# Funciones

---

- La función de concatenación o combinación, `c()`, es una de las funciones más utilizadas en *R*.
- 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)
> x
[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)
> x
[1] 1.0 1.5 2.0 2.5 3.0
```

# c()

---

- También puede usarse con caracteres:

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

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

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

# seq()

- Esta función permite formar sucesiones regulares de números. Su sintaxis es

```
seq(límite inferior, límite 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 1.0. Si no se especifica el límite inferior, el programa también asume el valor 1.0.

```
> 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
```

```
> b
```

```
[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 ()

- 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('Mé','xi','co'),3)
[1] "Mé" "xi" "co" "Mé" "xi" "co" "Mé" "xi" "co"
> 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 todo tipo de 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
```

## Ejercicio 5

---

1

Crea los siguientes vectores:

a) 15 2 56 21 45 2 7 "hola"

b) TRUE FALSE FALSE FALSE TRUE TRUE FALSE

2

Usando las instrucciones seq y rep genere las siguientes sucesiones:

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

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

# Funciones Estándar

---

Hay varias funciones para convertir números decimales a enteros:

- `round` Sintaxis: `round(x, n)`, donde `n` es el número de decimales. Números negativas redondean a potencias de 10.
- `trunc` Redondea al entero más cercano en la dirección al cero.
- `floor` Redondea al entero menor más cercano.
- `ceiling` Redondea al entero mayor más cercano.

```
> round(12.345)
[1] 12
> round(12.345, 2)
[1] 12.35
```

# *Funciones Estándar*

---

```
> trunc(12.345)
[1] 12
> floor(12.345)
[1] 12
> ceiling(12.345)
[1] 13
> trunc(-12.345)
[1] -12
> floor(-12.345)
[1] -13
> ceiling(-12.345)
[1] -12
```

# Funciones Estándar

---

Las funciones numéricas comunes están disponibles:

- `abs`, `sign`, `log`, `log10`, `sqrt`, `exp`, `sin`, `cos`, `tan`, `asin`, `acos`, `atan`, `sinh`, `cosh`, `tanh`, `gamma`, `lgamma`
- `sum`, `prod` dan la suma y el producto de las componentes de un vector. Las versiones acumuladas de estas operaciones son `cumsum`, `cumprod`.
- `max`, `min` dan el mayor y menor valor de las componentes de un vector. Las versiones acumuladas de estas operaciones son `cummax`, `cummin`.

# Funciones Estándar

---

- `range(x)` da como resultado `(min(x), max(x))` para un vector `x`.
- `sort(x)` ordena el vector `x` de manera creciente.
- `rev(x)` coloca las componentes de un vector o lista en orden inverso.
- `duplicated(x)` produce un vector lógico con valor `T` cuando la componente de un vector es un valor repetido.
- `unique(x)` elimina los valores duplicados.
- `union`, `intersect`, `setdiff`, `is.element` ejecutan las operaciones de conjunto  $A \cup B$ ,  $A \cap B$ ,  $A - B$  y  $n \in A$ . Sus argumentos pueden ser vectores de cualquier tipo pero, como conjuntos, no debe haber elementos repetidos.

# Funciones Estándar

---

- La función `% / %` denota la división entera. El resultado de `a % / %b` es `floor(a/b)`. Por ejemplo, la división entera de 3 entre 2 da como resultado 1:

```
> 3 %n % 2
[1] 1
> floor(3/2)
[1] 1
```

# Funciones Estándar

---

- Estas operaciones pueden usarse, por ejemplo, para identificar los múltiplos de cierto entero  $n$ , ya que en este caso el resto de la división entera debe ser 0.
- Por ejemplo, si  $x$  es un vector de enteros, los múltiplos de 3 son aquellas componentes que satisfacen la condición de que el resto es 0 al realizar la división entera por tres, es decir  $x \% 3 == 0$ :

```
> x <- 1:10
> x[x %% 3 == 0]
[1] 3 6 9
```

# *Expresiones Lógicas*

---

Si  $A$  y  $B$  son expresiones lógicas con valores vectoriales, entonces

- $A \ \& \ B$  es su intersección,
- $A \ | \ B$  es su unión ( $A$  o  $B$  o ambos),
- $\text{xor}(A, B)$  es su unión excluyente ( $A$  o  $B$  pero no ambos),
- $!A$  es la negación de  $A$ .
- Estas operaciones se efectuan de manera separada para cada componente de los vectores.

# *Expresiones Lógicas*

---

```
> x <- 1:6
> x > 2 & x <= 4
[1] FALSE FALSE TRUE TRUE FALSE FALSE
> x > 2 | x <= 4
[1] TRUE TRUE TRUE TRUE TRUE TRUE
> x <= 2 | x > 4
[1] TRUE TRUE FALSE FALSE TRUE TRUE
> x <= 2 & x > 4
[1] FALSE FALSE FALSE FALSE FALSE FALSE
> xor(x > 2 , x < 4)
[1] TRUE TRUE FALSE TRUE TRUE TRUE
> ! x<2
[1] FALSE TRUE TRUE TRUE TRUE TRUE
```

# Operadores lógicos

- 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 que (comparación) |
| !=      | No es igual a           |

**Tabla 4. Operadores lógicos.**

## Ejercicio 6

- 1 Genera un vector de 100 números aleatorios, usando la siguiente instrucción:

```
a <- round(runif(100, 0, 50))
```

- 2 Escribe una instrucción que indique cuando un elemento está en el conjunto  $b = \{2,3,6,7,8,19,22,27,31,25,37,39,44\}$ .

- 3 Para el vector  $a$  que generaste en la primera pregunta, cuenta cuántos elementos son iguales a 0.

- 4 Escribe una instrucción que cuente cuantos elementos del vector  $a$  son menores a 20.

# 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 letras escribimos:

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

# *El Operador Índice*

---

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

```
> 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 Índice*

---

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

Ejemplo: Asignamos a la variable 'a' los enteros del 1 al 26 por la asignación `a <- 1:26`. 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. Finalmente, usamos la expresión `a[a < 13]` como índice para obtener el subconjunto de los 12 primeros números.

# which()

- Funciona de forma similar al operador índice y sirve para hacer una lista de los elementos que satisfacen cierta condición. En este ejemplo, usaremos un conjunto de datos preexistente en *R*, denominado *trees*:

```
> trees
 Girth Height Volume
1 8.3 70 10.3
2 8.6 65 10.3
3 8.8 63 10.2
4 10.5 72 16.4
5 10.7 81 18.8
6 10.8 83 19.7
7 11.0 66 15.6
8 11.0 75 18.2
9 11.1 80 22.6
10 11.2 75 19.9
...
...
...
...
```

# 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
```

## Ejercicio 7

---

- 1 Usando el conjunto de datos `equipaje` que creaste en el Ejercicio 4, obtén todos los elementos que están entre 5 y 8 libras.

# 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:

```
x <- round(runif(500, 0, 1000))
y <- round(runif(500, 0, 1000))
x
y
z <- log((x^2 + 2*y) / (x + y)^2)
```

# *Operaciones con vectores*

- Ya hemos explorado algunas 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 5. Funciones vectoriales.**

# *Operaciones con vectores*

---

- Otras funciones de uso común para vectores son las siguientes:

| 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 duplicaciones |

**Tabla 6. Otras Funciones vectoriales.**

# La Función `order`

---

- La función `sort` permite ordenar las componentes de un vector. Para ordenar estructuras más complejas como una matriz o cuadro de datos existe la función `order`.
- El resultado de usar esta función sobre un vector es una permutación que reacomoda el vector en orden ascendente o descendente, según la opción `decreasing` que por defecto es cierta. Veamos un ejemplo

```
> (x <- c(4:6,2,1,3,7))
[1] 4 5 6 2 1 3 7
> order(x)
[1] 5 4 6 1 2 3 7
```

- El resultado quiere decir que para ordenar el vector `x` hay que colocar en el primer lugar la componente 5, en segundo lugar la componente 4, en tercer lugar la componente 6, y así sucesivamente.

# La Función order

---

- Si queremos ordenar el vector  $x$  ejecutamos la instrucción

```
> x[order(x)]
[1] 1 2 3 4 5 6 7
```

## Ejercicio 8

- 1 Genera un vector de nombre `v1` cuyas componentes sean 100 números generados al azar, usando la función del Ejercicio 6.
- 2 Obtén el máximo, mínimo, media, mediana y rango de `v1`.
- 3 ¿Cuántos valores distintos hay en el vector?
- 4 ¿En qué posición del vector se encuentran el máximo y el mínimo?

# 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 automáticamente.

# 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 renglones o 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.

# Matrices

---

- Si quisiéramos llenar la matriz por filas, es necesario poner 'T' como valor de `byrow`:

```
> matrix(1:12, nrow=3, byrow = T)
 [,1] [,2] [,3] [,4]
[1,] 1 2 3 4
[2,] 5 6 7 8
[3,] 9 10 11 12
```

# Matrices

---

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

# Matrices

---

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

**Tabla 2 Funciones matriciales.**

## Ejercicio 9

---

- 1 Construye dos matrices A1 y A2 de dimensiones 4 x 4.
- 2 Hallar  $A1 + A2$  y  $A1 \cdot A2$  (El símbolo  $\cdot$  denota el producto término a término).
- 3 Aplica las funciones `dim()`, `t()` y `diag()`, a ambas matrices y observa su efecto.

# Matrices

---

- 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'):

```
> cbind(A, 4:6)
 [,1] [,2] [,3]
[1,] 1 2 4
[2,] 3 4 5
[3,] 5 6 6
> rbind(B, c(5, 7))
 [,1] [,2]
[1,] 0 6
[2,] 2 8
[3,] 4 10
[4,] 5 7
```

# Matrices

---

- Para insertar nombres de filas o de columnas, se usan las funciones `rownames()` y `colnames()`, respectivamente:

```
> rownames(A) <- c('R1', 'R2', 'R3')
> colnames(A) <- c('C1', 'C2')
> A
```

|    | C1 | C2 |
|----|----|----|
| R1 | 1  | 2  |
| R2 | 3  | 4  |
| R3 | 5  | 6  |

## Ejercicio 10

1

Construye una matriz con la siguiente información

|          | PIB  | Pob | Infl |
|----------|------|-----|------|
| Austria  | 208  | 8   | 2.4  |
| Francia  | 1432 | 61  | 1.7  |
| Alemania | 2112 | 82  | 2.0  |

2

Añade a la tabla los valores para Suiza

|       |     |   |     |
|-------|-----|---|-----|
| Suiza | 259 | 7 | 1.6 |
|-------|-----|---|-----|

# RStudio

---

- *RStudio* es un entorno de desarrollo integrado (IDE) para *R* (lenguaje de programación) . Incluye una consola, editor de sintaxis que apoya la ejecución de código, así como herramientas para el trazado, la depuración y la gestión del espacio de trabajo.
- *RStudio* es una interfaz de *R*. ¡No funcionará en un equipo si no está instalado *R*!
- Nada cambia respecto a lo que has aprendido, sólo el entorno.
- Abre *RStudio*, y desde ahí el script del curso (*guion.R*) para continuar...

# Cuadros de Datos (Data Frames)

---

- Los cuadros de datos permiten mezclar distintos Tipos y Estructura de Datos en un mismo arreglo, de modo que se puede tener acceso a los datos como en una matriz. La sintaxis es

```
> data.frame (datos1, datos2, ...)
```

donde los puntos suspensivos indican que pueden especificarse tantos conjuntos de datos como sea necesario.

# *Cuadros de Datos*

---

- Para ver un ejemplo de la utilidad de los cuadros de datos, abriremos una tabla con datos reales extraídos de la base de datos de la Encuesta Nacional sobre Uso del Tiempo (ENUT) 2014.
- **Actividades:**
  - 1) En la carpeta del curso, explora el contenido de la tabla `enut2014.csv` usando Excel. Cierra el archivo.
  - 2) Abre la tabla desde *R* usando la función `read.csv()`. Guarda la tabla (data frame) en un objeto llamado 'enut14'.
  - 3) Este cuadro de datos lo usaremos más adelante...

# *Cuadros de Datos*

---

- Podemos tener acceso a las variables individuales de un cuadro de datos usando dos notaciones distintas:

`enut2014$edad` o `enut2014["edad"]`

- Como podemos ver, el resultado es muy extenso para desplegarlo completamente en la pantalla de la consola, no obstante podemos hacer uso del operador índice para que nos muestre, por ejemplo, las primeras cien observaciones de la variable:

`enut2014$edad[1:100]`

- O bien,

`enut2014[1:100,"edad"]`

- Nótese que dentro del operador índice aparecen ahora dos términos.

# Cuadros de Datos

---

- Tanto para matrices como para cuadros de datos, el operador índice funciona en dos dimensiones:  
  objeto[renglones, columnas]
- Ejemplo: Tomamos los datos del Ejercicio 10, que guardaremos ahora como un data frame llamado *paises*:

```
> PIB <- c(208,1432,2112,259)
> Pob <- c(8,61,82,7)
> Infl <- c(2.4, 1.7, 2.0, 1.6)
> paises <- data.frame(cbind(PIB,Pob,Infl))
> rownames(paises) <- c('Austria', 'Francia', 'Alemania',
 'Suiza')
> paises
```

|          | PIB  | Pob | Infl |
|----------|------|-----|------|
| Austria  | 208  | 8   | 2.4  |
| Francia  | 1432 | 61  | 1.7  |
| Alemania | 2112 | 82  | 2.0  |
| Suiza    | 259  | 7   | 1.6  |

# *Cuadros de Datos*

---

- Ahora, aplicamos el operador índice para extraer el valor del PIB de Alemania:

```
> paises['Alemania', 'PIB']
[1] 2112
```

- La inflación de Suiza:

```
> paises['Suiza', 'Infl']
[1] 1.6
```

- O la población de Francia:

```
> paises['Francia', 'Pob']
[1] 61
```

- Igual que antes, el operador índice funciona también con números o vectores:

```
> paises[1:3, 'Pob']
[1] 8 61 82
```

# Cuadros de Datos

---

```
> paises[3,2]
```

```
[1] 82
```

- Si se omite el argumento del operador índice correspondiente a los renglones, *R* mostrará todos los renglones de la(s) columna(s) dada(s) en el segundo argumento:

```
> paises[, 'Infl']
```

```
[1] 2.4 1.7 2.0 1.6
```

```
> paises[,2]
```

```
[1] 8 61 82 7
```

```
> paises[,1:2]
```

|  | PIB | Pob |
|--|-----|-----|
|--|-----|-----|

|         |     |   |
|---------|-----|---|
| Austria | 208 | 8 |
|---------|-----|---|

|         |      |    |
|---------|------|----|
| Francia | 1432 | 61 |
|---------|------|----|

|          |      |    |
|----------|------|----|
| Alemania | 2112 | 82 |
|----------|------|----|

|       |     |   |
|-------|-----|---|
| Suiza | 259 | 7 |
|-------|-----|---|

# *Cuadros de Datos*

---

- Del mismo modo, si se omite el argumento del operador índice correspondiente a las columnas, *R* mostrará todos las columnas de los renglones dados en el primer argumento:

```
> paises['Alemania',]
 PIB Pob Infl
Alemania 2112 82 2
```

```
> paises[c(2,4),]
 PIB Pob Infl
Francia 1432 61 1.7
Suiza 259 7 1.6
```

# *Cuadros de Datos*

---

- Como hemos visto antes, una ventaja de usar el operador índice es que funciona también bajo expresiones lógicas:

```
> paises[Infl < 2,]
 PIB Pob Infl
Francia 1432 61 1.7
Suiza 259 7 1.6
```

```
> paises[Infl < 2,1:2]
 PIB Pob
Francia 1432 61
Suiza 259 7
```

# Cuadros de Datos

---

- Como antes mencionamos, podemos invocar a las variables individuales de un cuadro de datos usando dos notaciones distintas:

`paises$Pob` o `paises["Pob"]`

- No obstante, usar esta notación puede resultar larga y poco práctica. Para ello, hay una alternativa que resulta útil cuando vamos a usar el mismo conjunto de datos repetidas veces: usar el comando `attach()` ('pegar').
- Si se usa este comando con un cuadro de datos, es posible trabajar usando sólo los nombres de las variables, sin hacer referencia al cuadro de datos en uso, hasta que se separe usando el comando `detach()`.
- Al usar el comando `attach()` hay que tener en cuenta que si existen variables fuera del cuadro de datos que tengan el mismo nombre que las variables del mismo, puede haber conflictos.

# *Cuadros de Datos*

---

- Veamos un ejemplo

```
> Pob <- 110 #Población de México
> Pob
[1] 110
attach(países)
The following object(s) are masked _by_.GlobalEnv :
 Infl, PIB, Pob
> Infl
[1] 2.4 1.7 2.0 1.6
> PIB
[1] 208 1432 2112 259
> Pob
[1] 110
> países$Pob
[1] 8 61 82 7
```

# *Cuadros de Datos*

---

- Observe que al dar la instrucción `attach`, *R* emite un alerta indicando que el objeto `Pob` esta 'cubierto' por otro en el ambiente de trabajo, y al pedir `Pob` el programa no nos da como respuesta los valores de esta variable en el cuadro de datos.
- La respuesta del programa es la variable local '`Pob`', que construimos previamente a su inclusión en el cuadro de datos.
- Para ver la variable del cuadro de datos podemos separar (`detach`) el cuadro, eliminar (`rm`) la variable `Fusion` y volver a pegar (`attach`) el cuadro de datos.

# *Cuadros de Datos*

---

- Solucionamos haciendo:

```
> detach(paises)
> PobMex2010 <- Pob
> rm(Pob)
> attach(paises)
```

The following objects are masked by .GlobalEnv:

Infl, PIB

```
> Pob
[1] 8 61 82 7
detach(paises)
```

# *Cuadros de Datos*

---

- Conclusión: La versión local de una variable precede (tiene preferencia sobre) a la versión que viene a través de la función `attach()`.

# *Cuadros de Datos*

---

- La función `attributes` permite listar las características de cualquier objeto en *R*.

```
attributes(paises)
$names
[1] "PIB" "Pob" "Infl"
```

```
$row.names
[1] "Austria" "Francia" "Alemania" "Suiza"
```

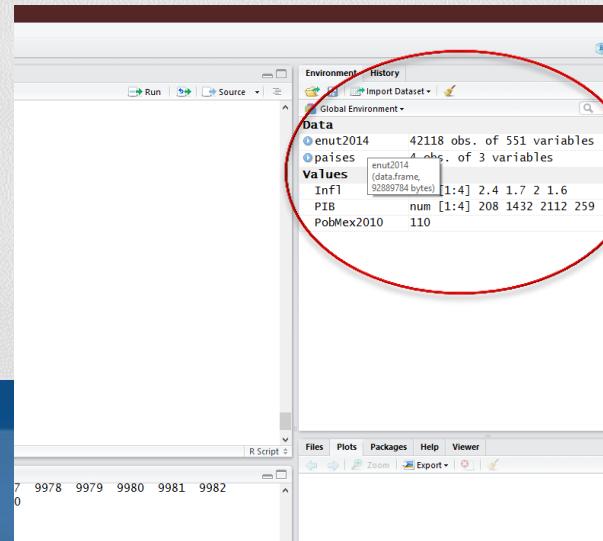
```
$class
[1] "data.frame">
```

# Cuadros de Datos

- La función `str` muestra la estructura del conjunto de datos.

```
> str(paises)
'data.frame': 4 obs. of 3 variables:
 $ PIB : num 208 1432 2112 259
 $ Pob : num 8 61 82 7
 $ Infl: num 2.4 1.7 2 1.6
```

- En Rstudio se puede obtener una vista de los datos de un cuadro haciendo clic en los objetos listados en la ventana superior derecha.



# Caso práctico con un data frame

---

- Pongamos en práctica ahora lo aprendido sobre data frames, usando datos reales.
- Usa las funciones `dim()`, `is.data.frame()`, `str()`, `names()` y `summary()` sobre el data frame y observa el resultado en cada caso.
- Limpia la consola de *RStudio*. Para hacerlo sitúa el cursor en ella y presiona *Ctrl+L*.
- Usa `attach()` para poder usar las variables de esta tabla como variables independientes.
- Obtén una vista de los datos del data frame `enut2014`, usando la interfaz de Rstudio.

# Caso práctico con un data frame

---

- Supongamos que queremos extraer las variables: edad, sexo y derechohabiencia de salud de los jefes del hogar. Como son demasiados datos, no los mostraremos en pantalla. Guardaremos esto en un objeto nuevo llamado “jef\_enut”:

```
attach(enut2014)
```

```
jef_enut <- enut2014[paren == 1 ,
c('edad', 'sexo', 'derh_sal')]
```

- Explora el resultado en los objetos de Rstudio.
- ¿Cuántas observaciones (jefes de hogar) quedaron en el nuevo objeto?
- Aplica las funciones `dim()`, `str()` y `names()` al data frame creado.

# Caso práctico con un data frame

---

- Sin embargo, puede ser que olvidáramos agregar alguna variable al nuevo data frame, quizá era necesario agregar las variables que registran el tiempo trabajado en la semana por esas personas, así como las relativas a su escolaridad.
- En el FD, podemos ver que dichas variables son: *p5\_3\_1* a *p5\_3\_4*, así como *niv* y *gra*.
- Ahora bien, cuando desplegamos el vector de nombres (usando `names(enut2014)`), podemos ubicar que las variables *niv* y *gra* ocupan las posiciones 8 y 9 de dicho vector, mientras que *p5\_3\_1*, ... *p5\_3\_4*, las posiciones 14 a 17. Lo anterior facilita la aplicación del operador índice, pues evita tener que referirse a dichas variables usando sus nombres entrecomillados. Para verlo claramente, definimos el vector de nombres acotado a esas variables:

```
names(enut2014) [c(8, 9, 14:17)]
```

# *Caso práctico con un data frame*

---

- Para agregar entonces las nuevas variables al data frame *jef\_enut* usamos por tanto, la función cbind:

```
jef_enut <- cbind (jef_enut, enut2014 [paren == 1 ,
c(8.9,14:17)])
```

y verificamos sus características con str:

```
str(jef_enut)
```

```
detach(enut2014)
```

# *Caso práctico con un data frame*

---

- De la estructura del data frame y observando el FD, podemos ver que la tabla contiene cuatro columnas asociadas al tiempo de trabajo:
  - p5\_3\_1 = Horas trabajadas de lunes a viernes
  - p5\_3\_2 = Minutos trabajados de lunes a viernes
  - p5\_3\_3 = Horas trabajadas de sábado a domingo
  - p5\_3\_4 = Minutos trabajados de sábado a domingo
- Sin embargo, resulta lógico pensar que se quiera tener una sola columna que contenga el total de horas semanales trabajadas, por lo que sería deseable agregar esto como una nueva variable al data frame

# Caso práctico con un data frame

---

- Para añadir una variable a un data frame, solamente hay que definirla usando la forma `data_frame$columna`. Si queremos usar ahora los nombres del data frame `jef_enut`, no olvidemos dar su respectivo `attach()`

```
attach(jef_enut)
```

```
jef_enut$hortrab <- p5_3_1 + p5_3_2/60 + p5_3_3 +
p5_3_4/60
```

- Observa que si tratamos de invocar la variable `hortrab` por su nombre, mostrando las primeras cien observaciones:
- `hortrab[1:100]`
- R devuelve un mensaje de error. Esto es porque no se actualizó el `attach()` para que incluya a la nueva variable.

# *Caso práctico con un data frame*

---

- Aplicamos entonces lo siguiente para actualizar:

```
detach(jef_enut)
```

```
attach(jef_enut)
```

```
hortrab [1:100)
```

- Con lo que ahora, podemos por ejemplo, graficar su curva de distribución:

```
plot(density(hortrab))
```

```
plot(density(hortrab, na.rm = T),
 xlab = "Horas",
 main = "Horas semanales de trabajo de los jefes del
 hogar")
```

# Función subset()

---

- En particular una función que resulta útil para manejar cuadros de datos y obtener subconjuntos de ellas es `subset()`.
- Subset() permite obtener un subconjunto de valores que satisfacen una condición dada. Por ejemplo, para extraer un cuadro de datos que contenga los valores para los hombres del data frame `jef_enut` podemos escribir:

```
> jef_hombr <- subset(jef_enut, sexo == 1)
```

## Ejercicio 11

- 1 Usa la función `hist()` sobre la variable de derechohabiencia de servicios de salud, `derh_sal`, para observar gráficamente como se distribuye entre los jefes del hogar.
- 2 Repite el procedimiento para las variables `sexo` y `niv`.
- 3 Calcula el promedio de edad de los jefes del hogar en la muestra.
- 4 Crea un nuevo data frame llamado `muj_jef`, que contenga las mismas columnas que `jef_enut` (excepto `sexo`), y que corresponda a las observaciones de la mujeres jefas del hogar.
- 5 Repite los histogramas de los puntos 1 y 2, así como el cálculo del promedio de edad del punto 3, para este nuevo data frame.
- 6 Discute sobre lo observado.

# La Función apply

- Este comando aplica de manera sucesiva una función a cada fila (primera dimensión), columna (segunda dimensión) o cada nivel de una dimensión superior. La sintaxis es

```
apply(datos, dim, función, ...)
```

- datos es el nombre de la matriz o arreglo y
- función es el nombre de cualquier función de R.
- Para una matriz (bidimensional), la opción dim puede tomar el valor 1 ó 2 para indicar las filas o las columnas, respectivamente.
- Los puntos suspensivos indican opciones que pueden ser necesarias para la función en cuestión.

# La Función apply

---

- Como ejemplo, podemos usar esta función para determinar el máximo de las variables de nuestro conjunto de datos *paises*.

```
> apply(paises, 2, max)
```

- La función `max` calcula el máximo para cada columna del conjunto de valores.

# La Función apply

---

- Para ver otro ejemplo del uso de esta función vamos a trabajar con el archivo `iris3` que tiene la información del conjunto de datos `iris` que hemos usado anteriormente pero en el formato de un arreglo de dimensiones  $50 \times 4 \times 3$ .
- La tercera dimensión corresponde a las especies, y para cada una de las tres especies hay una matriz de dimensión  $50 \times 4$  con los valores de las cuatro variables que ya conocemos para las 50 plantas de la especie correspondiente.

# La Función apply

---

- Para obtener el valor medio de cada variable para todo el conjunto de datos escribimos

```
> apply(iris3, 2, mean)
Sepal L. Sepal W. Petal L. Petal W.
5.843333 3.057333 3.758000 1.199333
```

# Conversión de Objetos

---

- Como hemos visto, las diferencias entre algunos tipos de objetos son pequeñas. Es posible convertir objetos de un tipo a otro cambiando algunos de sus atributos.
- Las instrucciones para este tipo de cambio tienen un formato común del tipo `as.algo`. *R* tiene más de 90 funciones de este tipo.
- El resultado de una conversión depende de los atributos del objeto.
- Algunos de los casos se resumen en la siguiente tabla.

# Conversión de Objetos

| Conversión a | Función      | Reglas                                                                                                   |
|--------------|--------------|----------------------------------------------------------------------------------------------------------|
| numérico     | as.numeric   | FALSE → 0<br>TRUE → 1<br>'1', '2', ... → 1,2,...<br>'A' → NA                                             |
| lógico       | as.logical   | 0 → FALSE<br>otros números → TRUE<br>'FALSE', 'F' → FALSE<br>'TRUE', 'T' → TRUE<br>otros caracteres → NA |
| carácter     | as.character | 1,2,... → '1', '2', ...<br>FALSE → 'FALSE'<br>TRUE → 'TRUE'                                              |

Tabla 1.9

# Datos Faltantes

---

- Para datos faltantes en un vector o cualquier otro objeto de *R* se usa el símbolo `NA`, que viene de las iniciales de *Not Available*, y es importante saber como reacciona *R* al encontrar un dato faltante.
- Cualquier operación aritmética que incluya valores `NA`, da como resultado `NA`. Por ejemplo

# Datos Faltantes

---

```
> aa <- c(1:3, , 9)
```

```
> aa <- c(1:3, NA, 9)
```

```
> aa
```

```
[1] 1 2 3 NA 9
```

```
> sum(aa)
```

```
[1] NA
```

```
> max(aa)
```

```
[1] NA
```

```
> 2*aa
```

```
[1] 2 4 6 NA 18
```

# Datos Faltantes

---

- Lo mismo ocurre con las relaciones `<`, `<=`, `>`, `>=`, `==`, `!=`.
- En particular, la expresión `x == NA` tiene como resultado `NA`

```
> aa == NA
[1] NA NA NA NA NA
```

- Sin embargo, el uso de los operadores lógicos de comparación usados con un vector que incluye valores `NA` da los siguientes resultados.

```
> aa>2
[1] FALSE FALSE TRUE NA TRUE
> aa[aa>2]
[1] 3 NA 9
```

# Datos Faltantes

---

- Para poder identificar las componentes de un vector (o de cualquier otro objeto) que son NA podemos usar la función `is.na()`, que da como resultado un vector lógico con valor TRUE cuando la componente es NA:

```
> is.na(aa)
[1] FALSE FALSE FALSE TRUE FALSE
```

- Con esta función podemos extraer las componentes que son NA:

```
> aa[is.na(aa)]
[1] NA
```

o las que no lo son:

```
> aa[!is.na(aa)]
[1] 1 2 3 9
```

# Datos Faltantes

---

- También podemos usar esta función para asignar el valor NA a una componente de un vector:

```
> (bb <- 1:9)
[1] 1 2 3 4 5 6 7 8 9
> is.na(bb) [6] <- T
> bb
[1] 1 2 3 4 5 NA 7 8 9
```

# Datos Faltantes

---

- Hay un segundo tipo de 'valores faltantes' que son el resultado de hacer un cálculo numérico cuyo resultado no es un número, y que se designan por las letras `NaN` (*Not a Number*). Por ejemplo

```
> 0/0
[1] NaN
> 1/0 + log(0)
[1] NaN
> is.na(0/0)
[1] TRUE
```

# Datos Faltantes

- Vemos que la función `is.na` identifica a `NaN` como un dato faltante. Para distinguir tenemos la función `is.nan()`. También existen las funciones `is.finite` y `is.infinite`. Veamos el efecto de estas funciones con un vector que tenga componentes de distintos tipos.

```
> (cc <- c(1, 1/0, 0/0, NA))
[1] 1 Inf NaN NA
> is.na(cc)
[1] FALSE FALSE TRUE TRUE
> is.finite(cc)
[1] TRUE FALSE FALSE FALSE
> is.infinite(cc)
[1] FALSE TRUE FALSE FALSE
> is.nan(cc)
[1] FALSE FALSE TRUE FALSE
```

# Manejo de Caracteres

---

- La función `nchar` da, en forma de vector, el número de caracteres en cada elemento de un vector de caracteres:

```
> (tt <- c('esta es una prueba', 'otra', 'y
 otra mas'))
[1] "esta es una prueba" "otra" "y otra mas"
> nchar(tt)
[1] 18 4 10
```

# Manejo de Caracteres

---

- La función `paste` usa un número arbitrario de argumentos y los une, elemento por elemento, produciendo un vector de caracteres. Por ejemplo

```
> paste(c('Altura','Peso'), rep(c(1,2),c(2,2)))
[1] "Altura 1" "Peso 1" "Altura 2" "Peso 2"
```

- Por defecto los elementos que se unen quedan separados por un espacio. Para evitar esto se puede usar el argumento `sep=algo`, donde algo es lo que se coloca entre los elementos, que puede ser incluso un espacio vacío. Por ejemplo,

```
> paste(c('X','Y'),1:4, sep=' ')
[1] "X1" "Y2" "X3" "Y4"
```

# Manejo de Caracteres

---

- El argumento `collapse`, permite que el resultado se concatene en una expresión larga, ya que permite determinar que carácter se coloca entre los componentes al hacer la concatenación. Por defecto toma el valor `NULL` y en consecuencia no se hace esta concatenación.

```
> paste(c('X','Y'),1:4, sep=",collapse='+')
[1] "X1+Y2+X3+Y4"
```

# Problema

## Ejercicio 12

1

Escriba las instrucciones para obtener un vector con las siguientes componentes

- [1] "Paciente 1 tiene altura igual a"
- [2] "Paciente 1 tiene peso igual a"
- [3] "Paciente 2 tiene altura igual a"
- [4] "Paciente 2 tiene peso igual a"

# Iteraciones

---

- Una iteración es un ciclo de operaciones que se repiten con cambios menores. En algunos lenguajes de programación la multiplicación de dos matrices requiere al menos tres ciclos encadenados.
- En *R* estas operaciones son mucho más sencillas de formular y también más eficientes. Siempre que sea posible, hay que tratar de evitar iteraciones.
- La gran mayoría de las veces una operación entre matrices es mucho más rápida. Trate de usar vectores y funciones como `apply`.

# Iteraciones

- Por ejemplo, definamos un vector con 50.000 componentes y calculemos el cuadrado de cada componente primero usando las propiedades de *R* de realizar cálculos componente a componente y luego usando un ciclo, y comparemos los tiempos que tarda cada una de estas operaciones.

```
> x <- 1:50000
> y <- x^2
> for (i in 1:50000) z[i] <- x[i]^2
```

- La segunda expresión no sólo es más larga sino que resulta en un cálculo mucho más lento que la primera.
- Sin embargo, hay situaciones en las cuales es imposible evitar el uso de ciclos. A continuación examinaremos las instrucciones de *R* que permiten construir ciclos en los programas.

# for

---

- La sintaxis de esta instrucción es

```
> for (i in ivalores) {instrucciones de R}
```

como en el siguiente ejemplo

```
> for (i in 1:10) {print (i)}
```

```
[1] 1
```

```
[1] 2
```

```
[1] 3
```

```
[1] 4
```

```
[1] 5
```

```
[1] 6
```

```
[1] 7
```

```
[1] 8
```

```
[1] 9
```

```
[1] 10
```

# for

---

- Si hay una sola instrucción, las llaves {} pueden omitirse.
- El objeto `i` valores puede ser de cualquier modo: un vector, una variable, una matriz, etc. *R* recorre todos los elementos del argumento reemplazando sucesivamente la variable `i` por los elementos.
- La variable `i` también puede tener cualquier modo: numérico, carácter, lógico o una combinación de estos.

```
> for (i in c(3,2,9,6)) {print (i^2)}
```

```
[1] 9
```

```
[1] 4
```

```
[1] 81
```

```
[1] 36
```

# while

---

- Si no se conoce el número de ciclos que se desea realizar antes de comenzar, se usa `while` que permite iterar hasta que cierto criterio se cumpla. Como ejemplo, vamos a sumar los enteros positivos hasta que la suma pase de 1000.

```
> n <- 0
> suma <- 0
> while (suma <= 1000)
{
 n <- n+1
 suma <- suma + n
}
> suma
[1] 1035
> n
[1] 45
```

- Vemos que el primer valor de la suma que pasa de 1000 es 1035 y que hicieron falta 45 iteraciones para llegar a este valor.

# Vectorización

- Dada la facilidad de trabajar con vectores en *R*, usualmente es posible reescribir bucles en términos de operaciones con vectores que resultan más breves y legibles.
- Veamos de nuevo el ejemplo de la suma de los enteros. Resulta más simple calcular 1000 valores y verificar en qué lugar se satisface el criterio de parada.

```
> n <- 1:1000
> su <- cumsum (n)
> su [su > 1000] [1]
[1] 1035
> n [su > 1000] [1]
[1] 45
```

- Para bucles grandes este enfoque es más rápido y más fácil de leer. Sin embargo es necesario tener una idea aproximada del número de iteraciones que se requieren.