Tutorial 02: Estadística descriptiva con R.

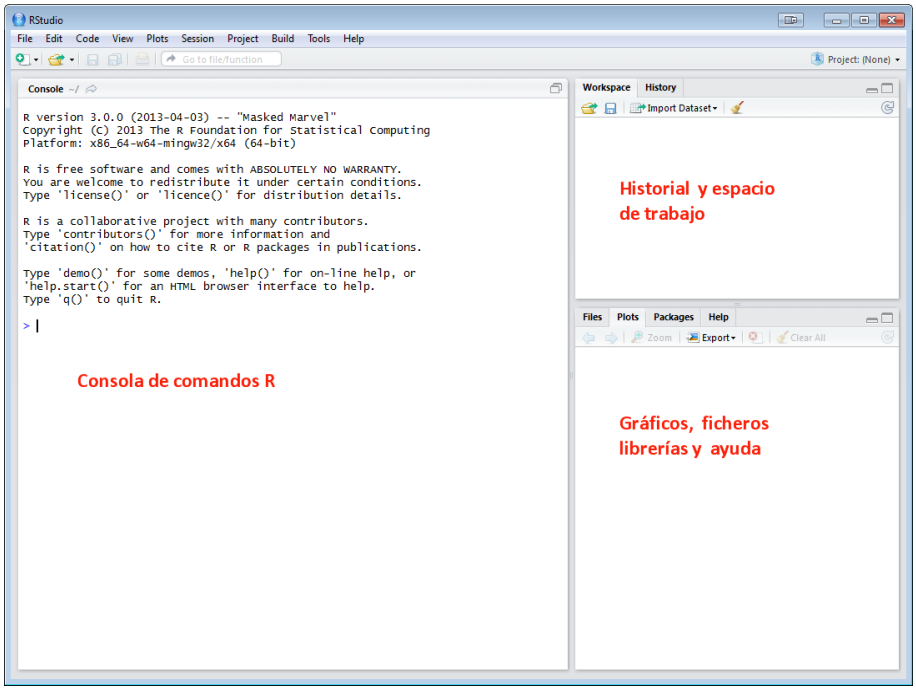
Atención:

- Este documento pdf lleva adjuntos algunos de los ficheros de datos necesarios. Y está pensado para trabajar con él directamente en tu ordenador. Al usarlo en la pantalla, si es necesario, puedes aumentar alguna de las figuras para ver los detalles. Antes de imprimirlo, piensa si es necesario. Los árboles y nosotros te lo agradeceremos.
- Fecha: 19 de septiembre de 2016. Si este fichero tiene más de un año, puede resultar obsoleto. Busca si existe una versión más reciente.

Índice

1. Primer contacto con R y RStudio.

Aunque la hoja de cálculo nos va a seguir acompañando durante todo el curso, nuestra herramienta fundamental va a ser R. En esta sesión vamos a familiarizarnos con RStudio, y aprenderemos a cargar un fichero de comandos R *"prefabricado"*. No te preocupes si al principio no entiendes algunas cosas, y otras tantas te parecen extrañas. Pronto irá quedando todo más claro.
Para empezar, abriremos el programa RStudio que instalamos en el Tutorial02. Cuando, tras instalarlo, iniciamos RStudio por primera vez, nos encontramos con una ventana similar a esta:



Como ves, la ventana aparece dividida en tres grandes regiones o *paneles*, que hemos rotulado en rojo en la figura:

- A la izquierda, la Consola de Comandos de R, en la que vamos a empezar nuestro trabajo.
- En la parte superior derecha, el panel con las pestañas de Historial (History),y Espacio de trabajo (Workspace) (en versiones recientes de RStudio en lugar del Espacio de Trabajo se muestra el Entorno (Environment); no te preocupes por esto de momento).
- En la parte inferior derecha, el panel con pestañas para `Ficheros (Files)`, `Gráficos (Plots)`, `Librerías (Packages)` y `Ayuda (Help)`. Las versiones recientes incluyen panel `Visor (Viewer)`.

En este y próximos tutoriales, y con la práctica, iremos conociendo la finalidad de cada una de esas regiones. Vamos a comenzar con la Consola de Comandos. En ella aparecen en primer lugar una serie de mensajes con información sobre la versión de R que estamos ejecutando (la 3.0.0 en esa figura, aunque seguramente tú habrás instalado una versión superior). Y bajo estos mensajes aparece el `prompt` de R, que es el símbolo `>`, junto al que parpadea el cursor.

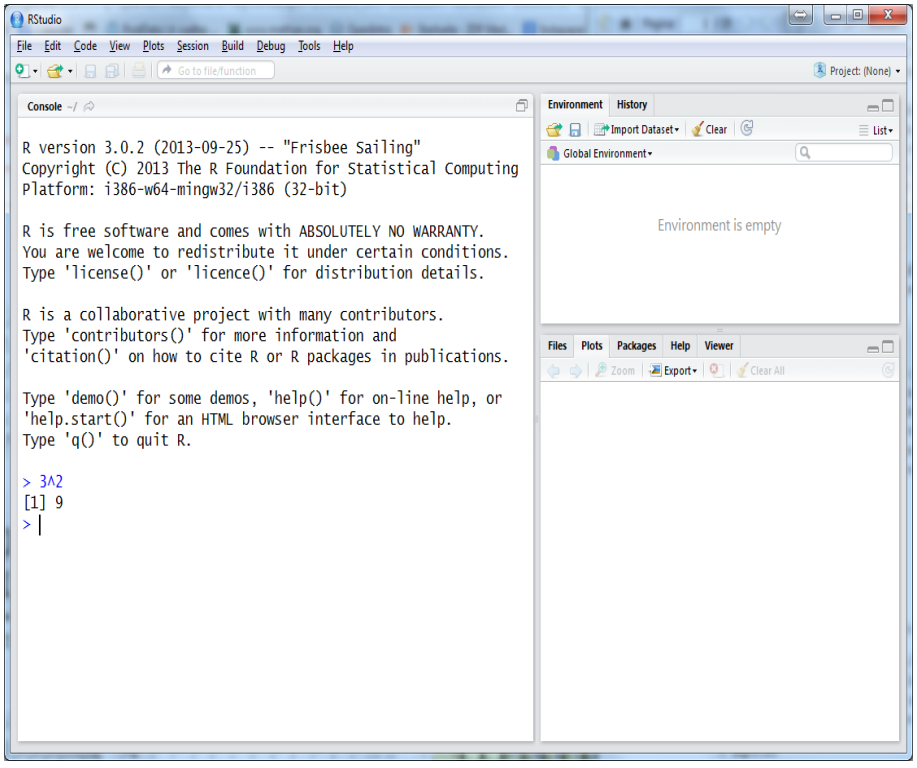
Ese símbolo, y el cursor parpadeando, nos indican que ese es el prompt activo, y que R está esperando una orden. En esta primera sección, vamos a aprender a usar R (a través de RStudio) como una calculadora. El objetivo principal es que empecemos a familiarizarnos con la sintaxis de R y con las peculiaridades de la interfaz de RStudio.
Empecemos el juego. Vamos a calcular 3^2. Asegúrate de que estás situado en la línea del prompt de R. Si es necesario, haz clic con el ratón junto al símbolo `>`, y después teclea:

```
3^2
```

El símbolo `^` es un acento circunflejo, el símbolo ∧, normalmente situado junto a la `P` de un teclado español estándar en Windows, y en los Mac, situado junto a la tecla `fn`. Al pulsar `Entrar` aparecerá una línea como esta:

```
[1] 9
```

y, justo debajo, aparece un nuevo prompt, que ahora pasa a ser el prompt activo.

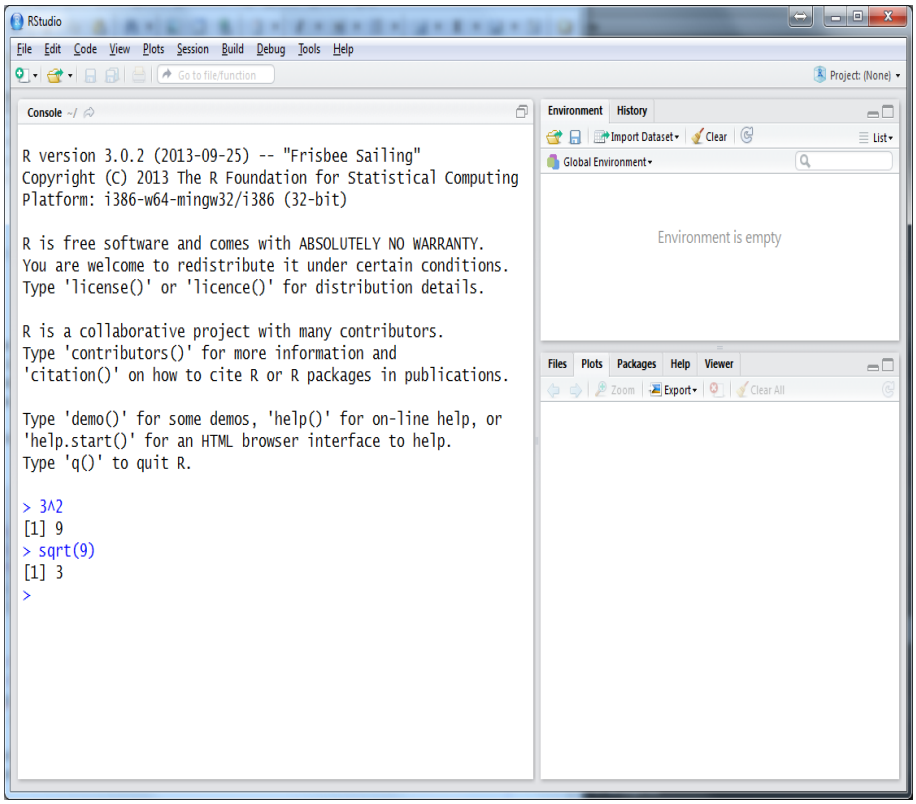


El uno entre corchetes es la forma en la que R nos indica que esta es la primera línea de su respuesta. En este caso la respuesta cabe holgadamente en una línea, pero pronto veremos casos en los que no es así. El número 9, naturalmente, es el valor 3^2 que hemos calculado. Como ves, en R se usa `^` para los exponentes, igual que en Calc.

Las operaciones aritméticas elementales se representan así en R (como en muchos otros programas):

- suma: `+`
- resta: `-`
- producto: `*`
- cociente: `/`
- potencia: `^`

La raíz cuadrada es una función en R, la primera que vamos a discutir. Se representa con `sqrt` (del inglés, *square root*). Es como en Calc, pero recuerda que, en R, la diferencia entre mayúsculas y minúsculas es, siempre, esencial; `Sqrt`, o `SQRT` no funcionarán. Se usa poniendo el argumento entre paréntesis. Por ejemplo, si tecleas `sqrt(9)` en el siguiente prompt activo, y lo ejecutas (pulsa `Entrar`) obtendrás el resultado previsible:



Un aviso: aunque hasta aquí te hemos mostrado el resultado mediante figuras, con capturas de pantalla, vamos a empezar a usar cada vez más descripciones *textuales* como esta:

```
> sqrt(9)
[1] 3
```

Sólo recurriremos a las capturas de pantalla en los casos en los que nos interesa destacar visualmente algún aspecto de la interfaz gráfica del programa. La ventaja (aparte de ahorrar espacio en el tutorial, claro) es que estas descripciones textuales te permitirán seleccionar el texto en la versión en pdf del tutorial, para usar *copiar y pegar* con RStudio. Ya irás viendo que esa forma de trabajar, aunque a veces tiene algún problema, a la larga te resulta cómoda. Pero, a la vez que te decimos esto, te recomendamos que, especialmente al principio, practiques tecleando tú mismo muchos de los comandos, para acostumbrarte a la interfaz, y para ayudarte a recordarlos mas eficazmente. En las descripciones textuales, a menudo te mostraremos juntos prompt, comandos y salida; en otros casos, nos limitaremos al comando R. Recuerda, si usas *copiar y pegar*, que el prompt `>` no forma parte del comando.
Naturalmente, además de `sqrt`, en R hay muchas otras funciones relacionadas con la Estadística que iremos encontrando en el resto del curso. Para practicar un poco más este uso de R como calculadora y la técnica de *copiar y pegar*, además de introducir un par de ideas nuevas, vamos a proponerte el primer ejercicio de este tutorial.

Ejercicio 1.
En este ejercicio vamos a utilizar las líneas de código que aparecen a continuación. Copia o teclea cada línea en el prompt (¡practica las dos maneras!), una por una, y ejecútalas, pulsando `Entrar` *tras copiar o teclear cada línea. Trata de adivinar el resultado de cada operación antes de ejecutar el código:*

```
2+3
15-7
4*6
13/5
13%/%5
13%%5
1/3+1/5
sqrt(25)
sqrt(26)
sin(pi)
sin(3.14)
```

□

Los resultados de este ejercicio aparecen en la Tabla 1 (pág. 5). ¡No los mires sin haberlo intentado! Esos resultados se merecen algunos comentarios:

- Seguramente, las líneas que menos previsibles te han resultado son las que contienen los operadores `%/%` y `%%`. Estos operadores representan, respectivamente, el cociente y el resto de la división entera.
- Lo más importante que tienes que observar en los resultados es que R trabaja *siempre* con la representación mediante decimales de los números. Esto es evidente, por ejemplo, en la respuesta a la instrucción

  ```
  1/3+1/5
  ```

  que representa la operación
  $$\frac{1}{3}+\frac{1}{5},$$
  pero cuyo resultado en R es `0.5333333` y no `8/15`. El valor decimal es una aproximación a la fracción, pero no es un resultado exacto (la fracción sí es exacta). Por lo tanto, cuando hagamos operaciones con R, tenemos que tener cuidado con la posible pérdida de precisión, que llevan siempre aparejadas las operaciones con decimales. Técnicamente, esto significa que R es un programa (esencialmente) numérico. Más adelante en el curso veremos algún programa simbólico, y discutiremos la diferencia entre ambos.
- Fíjate en que, como hemos dicho, R ha interpretado el símbolo

  ```
  1/3+1/5
  ```

  como la operación
  $$\frac{1}{3}+\frac{1}{5},$$
  en lugar de darle otras interpretaciones posibles como, por ejemplo:
  $$\frac{1}{\left(\frac{3+1}{5}\right)}.$$
  Para hacer esa interpretación R ha aplicado una serie de reglas, de lo que se conoce como prioridad de operadores, y que dicen en que orden se realizan las operaciones, según el tipo de operador. No queremos entretenernos con esto ahora, así que baste con decir que, en caso de duda, siempre puedes (y a menudo, debes) usar paréntesis para despejar la posible ambigüedad.

```
> 3^2
[1] 9
> sqrt(9)
[1] 3
> 2+3
[1] 5
> 15-7
[1] 8
> 4*6
[1] 24
> 13/5
[1] 2.6
> 13%/%5
[1] 2
> 13%%5
[1] 3
> 1/3+1/5
[1] 0.5333333
> sqrt(25)
[1] 5
> sqrt(26)
[1] 5.09902
> sin(pi)
[1] 1.224606e-16
> sin(3.14)
[1] 0.001592653
```

Tabla 1: Resultados del Ejercicio 1.

Por ejemplo, para distinguir entre las dos interpretaciones que hemos dado, puedes escribir:

```
(1/3)+(1/5)
```

o, por el contrario,

```
1 /( (3+1)/5 )
```

Un uso prudente de paréntesis y espacios en las operaciones es una marca característica del buen hacer, cuando se escribe código en un ordenador. Si alguna vez esto llega a ser un problema para ti, habrá llegado el momento de aprender algo más sobre esa *prioridad de operadores*.

- La función `sin` es la función *seno* de la Trigonometría. Además, R utiliza el símbolo `pi` para referirse a la constante matemática $\pi \approx 3.141593$. Y el carácter numérico de R se refleja, de nuevo, en el hecho de que al ejecutar `sin(pi)` no has obtenido 0 (que es la respuesta exacta), sino:

```
> sin(pi)
[1] 1.224606e-16
```

La notación que se usa en la respuesta es la forma típica de traducir la notación científica a los lenguajes de ordenador y calculadoras. El símbolo `1.224606e-16` representa el número:

$$1.224606 \cdot 10^{-16},$$

de manera que el número `-16`, que sigue a la letra `e` en esta representación, es el exponente de 10 (también llamado orden de magnitud), mientras que el número `1.224606` se denomina a veces mantisa. Puedes leer más sobre la notación científica en este artículo de la Wikipedia:

http://es.wikipedia.org/wiki/Notación_científica

En cualquier caso, el exponente `-16` nos indica que se trata de un número extremadamente cercano a 0. Fíjate, en cambio, que si usas 3.14 como aproximación de π, la respuesta, aunque pequeña, es todavía del orden de milésimas.

1.1. Algunos detalles de la interfaz de RStudio.

Antes de avanzar más, queremos ocuparnos de algunos aspectos del trabajo con RStudio que a menudo causan algunos quebraderos de cabeza a quienes se inician en el uso del programa.

Comandos incompletos

Para ver un ejemplo, prueba a escribir, en el prompt activo, esta operación evidentemente incompleta:

```
3*
```

y pulsa `Entrar`. La respuesta de R es un símbolo `+`:

```
> 3*
+
```

que en este caso no tiene nada que ver con la suma. Es la forma que tiene R de decirnos "necesito algo *más*". Cuando R se encuentra con una expresión que considera no errónea, sino incompleta (por ejemplo, con un paréntesis abierto al que corresponde un paréntesis cerrado), entonces nos lo comunica con este símbolo `+`. Y al hacerlo, nos da la oportunidad, en casos como este, de corregir el error, completando la expresión incompleta. Así que si, ahora, junto al símbolo `+` escribes 2, de esta forma:

```
> 3*
+ 2
```

y pulsas `Entrar`, verás que R ha entendido la expresión completa `3*2` y nso devuelve la respuesta:

```
> 3*
+ 2
[1] 6
```

En algún momento, no obstante, ese símbolo `+` de los comandos incompletos se convertirá en un inconveniente. Imagínate, por ejemplo, que querías multiplicar $5 \cdot 7$, pero que por error has tecleado la expresión

```
5*7-
```

y has pulsado `Entrar`, antes de darte cuenta del error. En tal caso, R piensa que esa expresión está incompleta, y vuelve a aparecer el signo `+`

```
> 5*7-
+
```

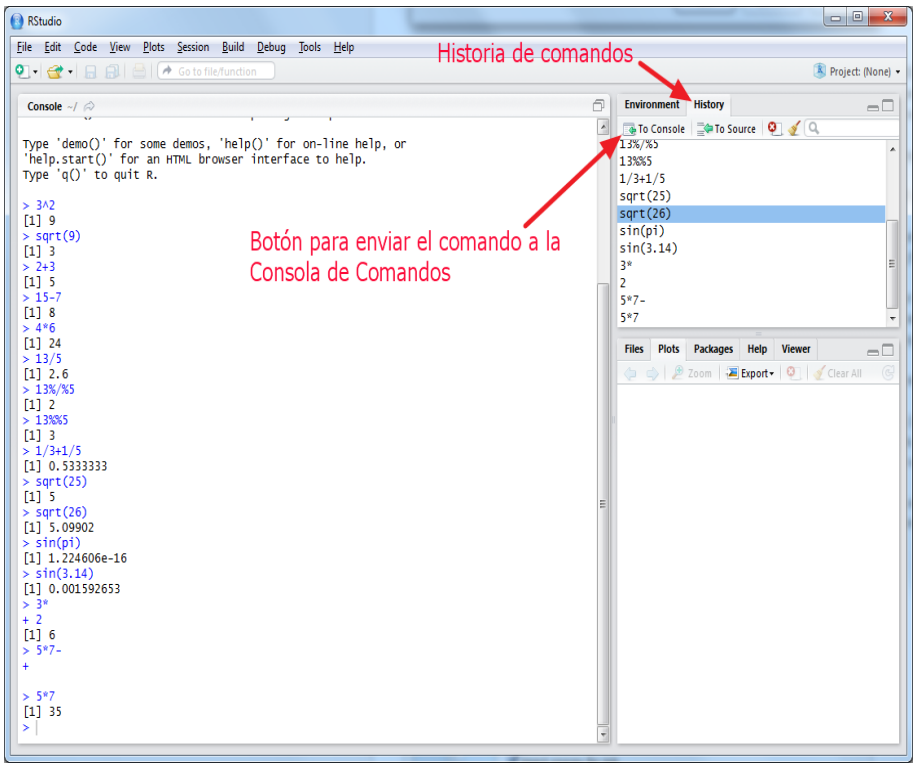
¡Pero ahora no queremos añadir nada! En este caso podrías añadir un 0, sin que eso afectara el resultado. pero habrá, más adelante, muchos casos en los que no hay una solución sencilla evidente. ¿Cómo salimos del paso? Simplemente, pulsamos la tecla `Esc` (escape), con lo que R interrumpe el comando y nos devuelve a un nuevo prompt activo, sin calcular nada.

Navegando el historial de comandos

En un caso como el que hemos descrito, no supone un problema volver a teclear la expresión, evitando el error que habíamos cometido. Pero en ocasiones, si hemos escrito una expresión realmente complicada, y, por ejemplo, hemos tecleado un paréntesis de más en medio de la expresión, seguramente preferiríamos no tener que volver a teclear la expresión completa. Un remedio "casero" consiste en seleccionar y copiar la expresión errónea, pegarla en el prompt activo, y editarla ahí para corregir el error.
Hay una segunda manera de conseguir el mismo efecto, Otra característica interesante de la Consola de Comandos es que, si te sitúas en el prompt activo, y pulsas las teclas con flechas arriba o abajo, verás pasar la lista completa de comandos que has ejecutado, desde el principio de la sesión de trabajo. Prueba a hacerlo ahora, para ver como funciona. Si has ejecutado todos los comandos que hemos ido describiendo en este tutorial, puedes pulsar la flecha ↑ repetidamente, e irás viendo pasar todos esos comandos, hasta llegar al

```
3^2
```

con el que hemos empezado. Y con la flecha hacia abajo recorres la lista en sentido contrario, hasta el comando de ejecución más reciente. Esa lista de comandos es el Historial de Comandos. Usando las flechas arriba y abajo puedes recorrer el Historial de Comandos, detenerte en cualquier punto, modificar ese comando si lo deseas, y después ejecutarlo (con o sin modificaciones). Al principio puede que te cueste un poco acostumbrarte, pero una vez que te habitúes a esta herramienta, la encontrarás, sin duda, muy útil. El Historial de Comandos que puedes recorrer con las flechas arriba y abajo se limita a la sesión de trabajo actual. Pero si miras en el panel superior derecho de RStudio, verás una pestaña titulada `History`, que se muestra en esta figura:



El Historial de Comandos que aparece en ese panel refleja tus comandos de esta y anteriores sesiones. Si esta es tu primera sesión de trabajo con R, ambos coincidirán, claro está. Pero a medida que vayas abriendo RStudio en días sucesivos, te puede resultar muy útil localizar, por ejemplo, un comando que usaste hace dos sesiones, pero que no recuerdas exactamente. Los comandos que aparecen en ese panel se pueden copiar y pegar en la Consola de Comandos o, de forma más directa, puedes seleccionar uno de ellos con el ratón, y pulsar en el botón `To Console` que hemos destacado en la Figura.

1.1.1. Errores.

Los errores son inevitables, y nos conviene estar preparados para su aparición. Dependiendo del motivo del error, R nos informará con distintos tipos de mensajes. Así que vamos a ver algunos ejemplos, para conocer la terminología asociada.
Para empezar, este es el mensaje de error que obtenemos tras ejecutar una expresión sin sentido:

```
> 4/*3
Error: unexpected '*' in "4/*"
```

Los mensajes de error aparecen en color rojo en RStudio. Este mensaje de error en particular, es bastante fácil de entender: R simplemente se queja de que esta expresión es defectuosa. No incompleta, como alguna que hemos visto antes. Es imposible completar esta expresión de forma correcta.
Veamos otro tipo de mensaje de error. Ya sabes que no se puede dividir por 0. Si lo intentamos en R

```
> 2/0
[1] Inf
```

el símbolo `Inf` (de infinito) nos avisa del problema. Ese símbolo también puede aparecer con signo menos, si por ejemplo tratas de calcular el logaritmo neperiano de 0. En R el logaritmo neperiano es la función `log`, así que sería:

```
> log(0)
[1] -Inf
```

En ambos casos R no lanza un mensaje de error, pero utiliza el símbolo de infinito, porque así nos proporciona más información matemática sobre lo que ha ocurrido. También puede suceder que el símbolo `Inf` aparezca como *resultado intermedio* en una expresión. En ese caso R aplica unas reglas sencillas de *aritmética con infinitos*, para tratar de darnos la respuesta más coherente desde el punto de vista matemático. Por ejemplo, si calculas

```
3*(-5/(4-2^2))
```

Entonces R empieza calculando el resultado del paréntesis `4-2^2`, que es 0. Así que al dividir -5 entre 0, R obtiene `-Inf`, y finalmente, al calcular `3*(-Inf)`, se obtiene `-Inf` como valor de la expresión. Comprobémoslo:

```
> 3*(-5/(4-2^2))
[1] -Inf
```

A pesar de que R no lanza mensajes de error, está claro que, en la mayoría de los casos, no es deseable que nuestros cálculos produzcan `Inf` como resultado, en ninguna etapa.
En otros casos, el problema es diferente. Por ejemplo, si tratas de calcular la raíz cuadrada de un número negativo:

```
> sqrt(-4)
[1] NaN
Warning message:
In sqrt(-4) : NaNs produced
```

En este caso obtenemos una respuesta `NaN`, y además una advertencia (*warning*) sobre la aparición de esa respuesta. El símbolo `NaN` es la abreviatura de *Not a Number*; es decir "no es un número". Esta es la respuesta que obtendremos cuando tratamos de hacer operaciones como esta, que producen como resultado un número complejo. R es capaz de trabajar con los números complejos, pero como no los vamos a necesitar en el curso, no nos ocuparemos de esto. Para que veas otro ejemplo, esa respuesta `NaN` aparece también si tratamos de ir demasiado lejos con la *aritmética con infinitos:*

```
> (1/0)+log(0)
[1] NaN
```

En este caso R se ha encontrado en un paso intermedio con `Inf - Inf`, y su respuesta ha sido `NaN`.

En cualquier caso, y como resumen final, tenemos que tratar de evitar que nuestros cálculos conduzcan a este tipo de expresiones.

Haciendo limpieza en la Consola de Comandos.

Si has seguido todas las instrucciones hasta aquí, tu Consola de Comandos seguramente empieza a estar llena de comandos, y algunos pueden quedar ocultos en la parte superior. Naturalmente, puedes usar las barras de desplazamiento para moverte por la Consola. Pero en sesiones largas de trabajo, o cuando hemos terminado de hacer pruebas y queremos ejecutar una versión definitiva de nuestro trabajo, es normal que queramos hacer limpieza en la Consola. Eso se consigue con la combinación de teclas `Ctrl + L`. ¡Cuidado! Es un proceso no reversible: podrás seguir navegando el Historial de Comandos con las teclas arriba y abajo, pero para recuperar los resultados tendrás que ejecutar de nuevo los comandos.
Más adelante aprenderemos formas de guardar nuestro trabajo de una manera más eficiente. Pero te podemos adelantar que, puesto R se maneja *escribiendo comandos*, bastará con guardar esos comandos en ficheros de texto plano. Ahora vamos a cambiar de tema, y para empezar desde cero, vamos a cerrar RStudio para eliminar las trazas del trabajo que hemos hecho.

2. Variables y vectores.

Vuelve a abrir RStudio, para continuar el trabajo. En la sección previa hemos usado R como una calculadora. Pero, para ir más allá, tenemos que disponer de *estructuras de datos*. Ese término

describe, en Computación, las herramientas que nos permiten almacenar y procesar información. Las estructuras de datos más básicas de R son las variables y los vectores. En próximos tutoriales nos iremos encontrando con otras estructuras de datos: matrices, data.frames, listas, entre otras.

2.1. Variables en R.

Una variable, en R, es un símbolo que usamos para referirnos a un valor. Por ejemplo, si te sitúas en la Consola de Comandos, tecleas

```
a = 2
```

y ejecutas esa instrucción, entonces, aunque aparentemente no ha sucedido nada (porque no hay respuesta), a partir de ese momento R ha asignado el valor 2 al símbolo `a`. Así que si, por ejemplo, tecleas y ejecutas

```
a + 1
```

obtendrás como respuesta 3.

Ejercicio 2.
¿Qué se obtiene al ejecutar estos comandos, uno detrás de otro? ¿Cuánto valen las variables `a`*,* `b` *y* `c` *al final?*

```
a = 2
b = 3
c = a + b
a = b * c
b = (c - a)^2
c = a * b
```

La solución está en la página 48. ¡No la mires antes de intentar pensar el ejercicio por ti mismo! □

En el Capítulo 1 del libro hemos hablado de variables cualitativas y cuantitativas. Estas últimas toman siempre valores numéricos, y las variables de R sirven, desde luego, para almacenar esa clase de valores. Pero, como iremos viendo en sucesivos tutoriales, también se pueden utilizar variables de R para guardar valores de variables cualitativas (factores), y otros tipos de objetos que iremos conociendo a lo largo del curso. De momento, para que veas a que nos referimos, recordaremos el Ejemplo 1.1.1 del libro (pág. 6), en el que teníamos una variable cualitativa ordenada que representa el pronóstico de un paciente que ingresa en un hospital. Prueba a ejecutar este código:

```
pronostico = "Leve"
```

simplemente para que, de momento, veas que:

- R no protesta. El valor `"Leve"` es un valor perfectamente aceptable.
- En R los valores que representan palabras, frases, o como las llamaremos técnicamente cadenas alfanuméricas, se escriben siempre entre comillas. Puedes usar comillas simples o dobles, pero te aconsejamos que uses dobles comillas.

La relación entre factores y valores alfanuméricos en R es algo más complicada de lo que este sencillo ejemplo parece sugerir. Pero ya tendremos ocasión, en futuros tutoriales, de extendernos sobre eso.

Asignaciones

Las instrucciones como

```
a = 2
```

o como

```
b = (c - a)^2
```

se denominan `asignaciones`. En la primera, por ejemplo, decimos que hemos asignado el valor `2` a la variable `a`.
Queremos advertir al lector de que en R hay otra forma, esencialmente equivalente, de escribir las instrucciones, y que, de hecho, se usa mayoritariamente. En lugar del símbolo `=`, puedes conseguir el mismo efecto escribiendo `<-` (el signo "menor" seguido, sin espacios, de un signo "menos"). Así que, si ejecutas

```
a <- 2
```

el resultado será el mismo que antes con `=`. Las dos notaciones tienen ventajas e inconvenientes, pero nosotros preferimos `=` porque se parece más a lo que se utiliza en muchos lenguajes de programación.

Cómo ver los resultados de un cálculo a la vez que lo hacemos

Ya habrás notado que, al hacer una asignación, R no produce ninguna respuesta, y se limita a guardar el valor en la variable. En una asignación como `a=2`, esto no genera ninguna duda. Pero si el miembro derecho de la asignación es una expresión más complicada, puede ser conveniente pedirle a R que nos muestre el resultado de esa expresión *a la vez* que se hace la asignación. Imagínate, por ejemplo, esta situación:

```
b =  2
c = -2
d = -4
a = b * c^2 / d + 3
```

¿Cuál es el valor que se asigna a la variable `a`? Una forma de averiguarlo es, simplemente, ejecutar una línea adicional, en la que aparece sólo el nombre de la variable que queremos inspeccionar. Al ejecutar esta sentencia, R nos devuelve como respuesta:

```
> a
[1] 1
```

Pero hay otra manera de conseguir que R nos muestre el resultado de la asignación que es más interesante, porque produce código más simple. Basta con encerrar toda la asignación entre paréntesis. El resultado es:

```
> ( a = b * c^2 / d + 3 )
[1] 1
```

como antes. Este ejemplo es, desde luego, bastante artificial. Te recordamos, en cualquier caso, la conveniencia de utilizar paréntesis para hacer más legibles las expresiones, y evitar posibles errores. Pero, a la vez, la posibilidad de comprobar el resultado de las asignaciones, aumenta la fiabilidad del código que escribimos.

Nombres de las variables

Aunque hasta ahora hemos usado letras como nombres de las variables, puedes utilizar nombres más descriptivos. Y muchas veces es una buena idea hacerlo. Por ejemplo, puede que hace una semana hayas escrito estas instrucciones para resolver un problema:

```
a = 2
b = 3
c = a / b
```

Pero si las vuelves a ver, pasada una semana, es posible que no recuerdes qué era lo que estabas tratando de conseguir al hacer esto. En cambio, al ver estas instrucciones:

```
espacio = 2
tiempo = 3
velocidad = espacio / tiempo
```

es mucho más fácil reconocer el objetivo que persiguen. A lo largo de los tutoriales del curso vamos a insistir muchas veces en la necesidad de que el código esté bien *organizado*, y esté bien *documentado*. Un primer paso en esa dirección es tratar de elegir nombres descriptivos para las variables. En R las reglas para los nombres de variables son muy flexibles: esencialmente, que empiecen por una letra y no contengan espacios ni caracteres especiales, como `?`, `+`, paréntesis, etcétera. Pero cuidado con los excesos. Es cierto que puedes usar nombres de variables arbitrariamente largos. Pero si usas como nombre:

```
Estavariablealmacenaelresultadodelaoperaciontaninteresantequeacabamosdehacer
```

tu trabajo resultará ilegible. Como siempre, se necesita un equilibrio, y con la práctica encontrarás el tuyo (consejo zen gratuito del tutorial de hoy). Para empezar, es una buena idea combinar mayúsculas y minúsculas en los nombres de las variables. Si vas a usar una variable para representar la temperatura final de un proceso, el nombre `temperaturafinalproceso` es menos legible y más largo que `tempFinal`. Es mucho más fácil, por otra parte, que te equivoques tecleando el primero de esos nombres.

2.2. Vectores.

En Estadística, lo habitual es trabajar con colecciones o *muestras* de datos. Y para almacenar esas colecciones de datos, la estructura más básica de R son los vectores. En este apartado vamos a empezar nuestro trabajo con ellos, aunque a lo largo del curso aún tendremos ocasión de aprender bastante más sobre el manejo de los vectores de R.
Para empezar vamos a trabajar con vectores que contienen una colección de números, que pueden ser las edades de los alumnos de una clase:

22, 21, 18, 19, 17, 21, 18, 20, 17, 18, 17, 22, 20, 19, 18, 19, 18, 22, 20, 19

En R, las listas de números como esta se almacenan en vectores. El vector que corresponde a esas edades se construye en R mediante un comando como este:

```
edades = c(22, 21, 18, 19, 17, 21, 18, 20, 17, 18, 17, 22, 20, 19, 18, 19, 18, 22, 20, 19)
```

En realidad, en esa expresión hemos hecho dos cosas:

1. Hemos *creado* el vector con los datos, en la parte derecha de la expresión:

   ```
   c(22, 21, 18, 19, 17, 21, 18, 20, 17, 18, 17, 22, 20, 19, 18, 19, 18, 22, 20, 19)
   ```

   Los datos están separados por comas, y rodeados por paréntesis. Hasta aquí todo es bastante típico, pero *fíjate especialmente en la letra* `c` que precede al primer paréntesis. Esa letra `c` es el nombre de una nueva función de R, y es la forma más sencilla que existe en R para crear un vector de datos. Podría pensarse que la c es de *crear*, pero en realidad proviene de *concatenar*. Enseguida entenderemos porque es así.

2. Una vez creado el vector, lo hemos *asignado* a la variable `edades`. Hasta ahora sólo habíamos usado variables para identificar un único valor (un número o una cadena alfanumérica). Pero una variable puede usarse para identificar un vector o, como veremos más adelante, estructuras mucho más complejas.

Una observación más: los vectores de R se usan para almacenar valores de una variable. Y en particular, todos los datos de un vector son del mismo tipo de variable. No se puede mezclar, en un vector, números y cadenas alfanuméricas (aparentemente R te dejará hacerlo, pero internamente habrá convertido todos los números en cadenas alfanuméricas).

Manos a la obra. Prueba a copiar el comando anterior en tu *Consola de comandos* (puedes usar *copiar y pegar* desde este fichero) y ejecútalo. Si las cosas van bien, no esperes ninguna respuesta: como ya hemos dicho, R no te muestra el resultado de las asignaciones. Recuerda que, si quieres ver el resultado de una asignación, puedes rodear toda línea con paréntesis, como en:

```
(edades = c(22, 21, 18, 19, 17, 21, 18, 20, 17, 18, 17, 22, 20, 19, 18, 19, 18, 22, 20, 19))
```

Otra opción, naturalmente, es preguntarle a R por el valor de `edades`, después de la asignación. Escribe simplemente `edades` en la *Consola de Comandos* y ejecútalo. Verás, de nuevo, el contenido de esa variable.

Vamos a hacer algunas operaciones con este vector. Imagínate que, como sucede a menudo, después de haber creado nuestra lista de edades de alumnos, desde la administración nos avisan de que hay cinco alumnos nuevos, recién matriculados, y debemos incorporar sus edades, que son

22, 18, 20, 21, 20

a nuestro vector de datos. Naturalmente, podemos empezar de nuevo, pero es preferible reutilizar el vector `edades` que ya habíamos creado. Vamos a ver esto como primer ejemplo, para empezar a aprender cómo se manipulan vectores en R. Una forma de añadir los datos es empezar creando un nuevo vector (con la función `c` de R):

```
(edades2 = c(22, 18, 20, 21, 20))
```

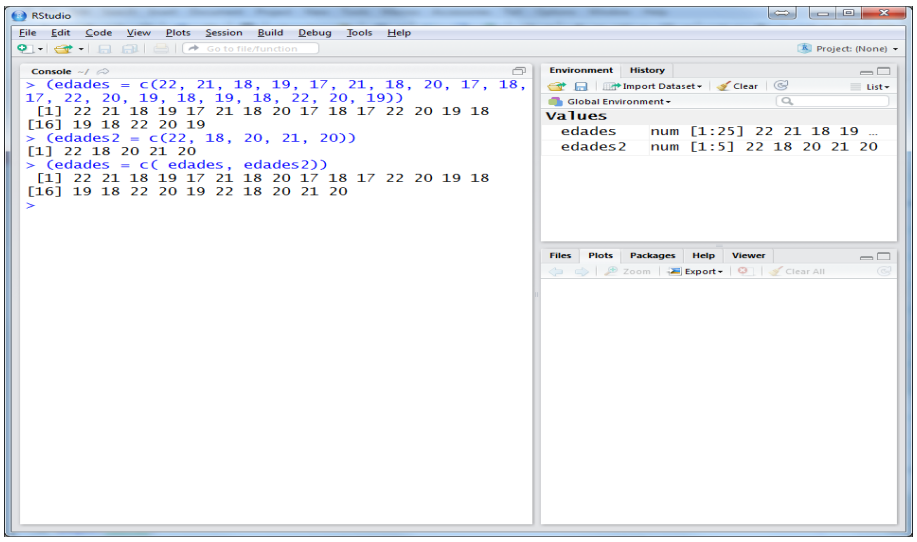
Y a continuación *concatenamos* los vectores, usando de nuevo la misma función `c`, de una manera distinta:

```
(edades = c( edades, edades2))
```

Hemos rodeado toda la línea con paréntesis, para ver el resultado y poder observar cómo ha cambiado el vector `edades`.
Vamos a analizar paso a paso lo que ha ocurrido:

- La instrucción `c( edades, edades2)` toma los datos contenidos en `edades` y `edades2` y los *concatena*, en ese orden, para fabricar un nuevo vector, que todavía no está guardado en ninguna variable.
- La asignación `edades = c( edades, edades2)` toma ese vector y lo guarda en la propia variable `edades`. Al hacer esto, como era de esperar, el anterior vector `edades` es reemplazado con la nueva lista, y su contenido anterior se pierde (cuidado con esto; las reasignaciones son una de las formas habituales de cometer errores).

El nombre de la función `c` proviene, en realidad, de *concatenate*, concatenar en inglés.
La siguiente figura resume lo que hemos venido haciendo con estos datos:



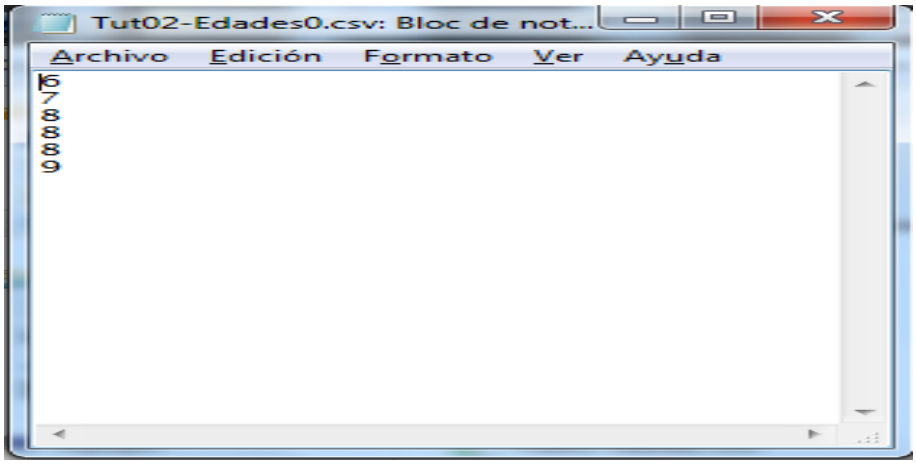
Los números entre corchetes, al principio de cada fila, como el `[1]` y el `[16]` de este ejemplo, son sólo ayudas para que podamos localizar los elementos del vector. En mi pantalla la primera fila muestra los primeros 15 números del vector. Pero en la tuya, si la ventana de RStudio es de distinta anchura, los números entre corchetes pueden ser distintos. Hasta ahora sólo habíamos visto ejemplos con `[1]`, que es lo que sucede cuando la respuesta es un vector que cabe en una única línea.

3. Ficheros `csv` con R.

En esta sección vamos a aprender a utilizar ficheros `csv` con R. En el Tutorial-01 hemos visto algunos ejemplos de ese tipo de ficheros, y el manejo básico con una hoja de cálculo, como Calc. Como vimos allí, un fichero `csv` típico contiene una tabla de datos con varias columnas. Esa estructura de tabla es la más habitual para intercambiar información y, como veremos más adelante, unas formas de organizar las tablas son más eficientes que otras (hablaremos de *tidy data*, en inglés, datos organizados). Durante buena parte del curso nos vamos a limitar a problemas en los que interviene una única variable, es decir, o bien los datos de interés están en una de las columnas de la tabla o bien se trata de una tabla con una única columna. Es el caso del siguiente fichero `csv` adjunto:

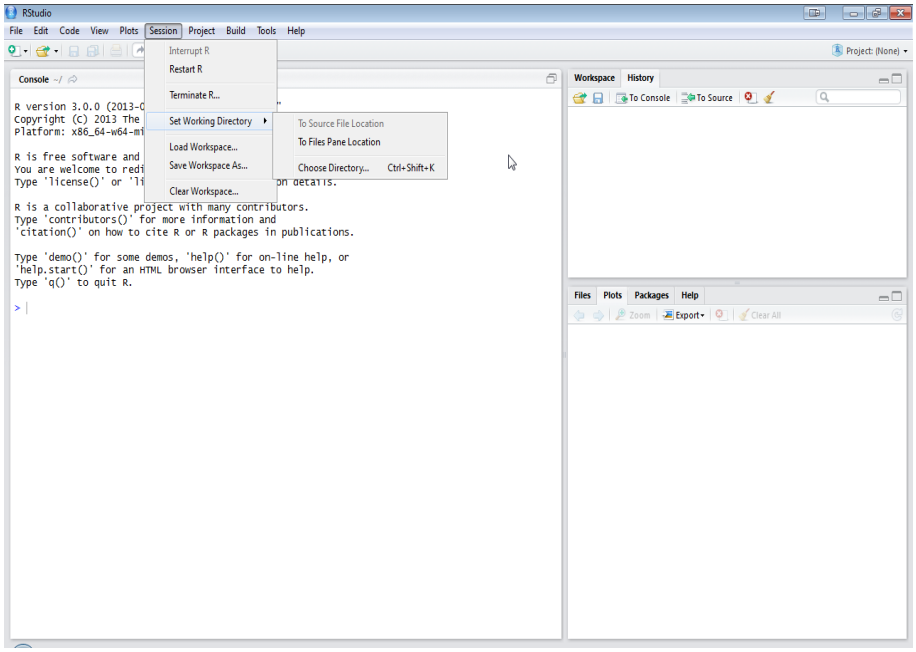
Tut02-Edades0.csv

Guárdalo en la subcarpeta `datos` de tu directorio de trabajo (el que contiene este pdf). Si lo abres con un editor de texto (como el *Bloc de Notas*, en Windows), verás, como muestra esta figura,



que el fichero contiene sólo una columna de datos, que en este ejemplo corresponde a valores de una variable cuantitativa discreta.
Queremos utilizar esos datos en R, y para eso vamos a (1) leerlos desde el fichero y (2)guardarlos en un vector. El resultado será como si nosotros hubiéramos creado ese vector tecleándo sus elementos directamente.
Para el primer paso (la lectura del fichero `csv`), vamos a tener que empezar por explicarle a R (usando RStudio como intermediario) dónde está el fichero que contiene esos datos. Cuando trabajamos con ficheros, R utiliza un directorio de nuestro ordenador como directorio de trabajo. A continuación vamos a aprender a elegir un directorio de trabajo. Puedes usar cualquier carpeta, y en lo que sigue vamos a suponer que hemos elegido el *Escritorio*. Lo más importante es que esa carpeta debe contener a su vez, una subcarpeta llamada `datos`. Y asegúrate de que los ficheros `csv` que vas a usar están almacenados en esa subcarpeta llamada `datos`.

Usamos el menú `Session` → `Set Working Directory` → `Choose Directory`.



Se abre un cuadro de diálogo del sistema, y navegamos hasta la carpeta que contiene el fichero `csv`, que en mi caso es el `Escritorio`. Cuando selecciono el Escritorio, veo aparecer en la Consola de Comandos de R, un comando, que RStudio ha escrito y ejecutado por nosotros:

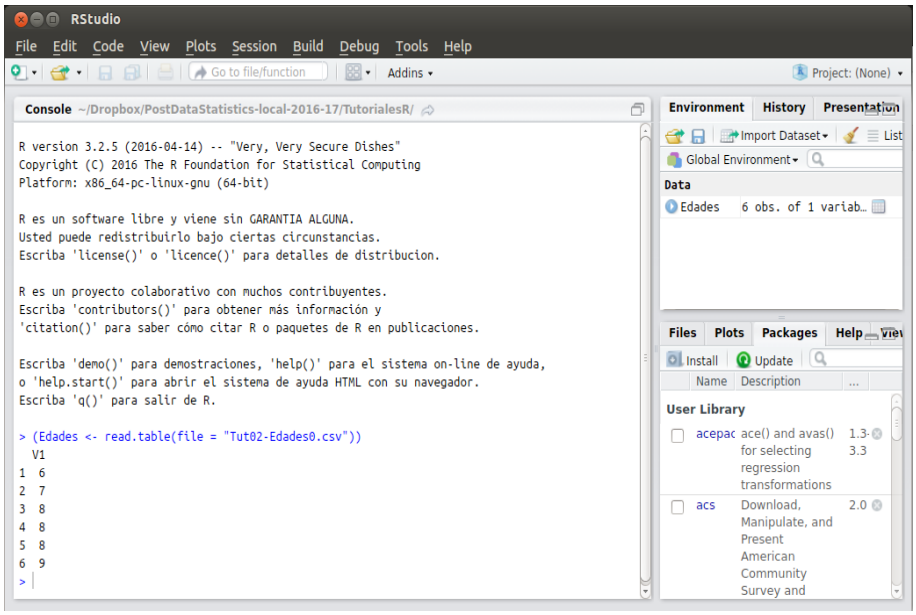
```
setwd("C:/Users/Fernando/Desktop")
```

Este es el resultado en mi ordenador (de ahí el nombre de usuario), usando Windows 7. En versiones más antiguas de Windows, o en una máquina con Linux, o un Mac, el comando será ligeramente distinto. Cuando tengamos más práctica, aprenderemos otras formas de seleccionar el directorio de trabajo, o incluso de utilizar ficheros situados fuera de ese directorio de trabajo. Pero, por el momento, resulta más cómodo dejar que RStudio haga ese trabajo para nosotros.

Ahora R ya sabe dónde está el fichero que nos interesa, pero todavía no sabe cual es. Para ello vamos a utilizar la función `read.table` de R. Aunque esta función tiene muchas opciones disponibles, nuestro primer uso de ella no podría ser más sencillo. Copia y ejecuta el siguiente código en RStudio:

```
( Edades = read.table("../datos/Tut02-Edades0.csv") )
```

En esta instrucción le decimos a R que use la función `read.table` para leer el contenido del fichero `Tut02-Edades.csv` y que almacene lo que ha leído en una variable, que hemos decidido llamar `Edades`. Al hacerlo verás una respuesta como esta



y hay varias cosas sobre las que detenerse:

- Hemos pedido a R que lea una tabla que tiene una única columna y, como no tenía nombre, le ha asignado uno `V1`. Veremos más adelante que podemos referirnos a las columnas por la posición que ocupan en la tabla (primera, segunda,...) o por su nombre.
- Los valores contenidos en la única columna de la tabla son `6, 7, 8, 8, 8, 9`
- Además, R enumera las filas, esos son los valores `1, 2,..., 6` que aparecen al principio de cada linea.

La variable `Edades` es una tabla. Podemos preguntar a R de qué clase (tipo) es esa variable con el comando `class`

```
> class(Edades)
[1] data.frame
```

Para guardar el contenido de esa columna en un vector, que llamaremos `vectorEdades0` usaremos el símbolo `$` entre el nombre de la tabla `Edades` y el de la columna `V1`:

```
> (vectorEdades0 <- Edades$V1)
[1] 6 7 8 8 8 9
```

Ahora podemos comprobar que tras la variable `vectorEdades0` hay, efectivamente, un vector de números enteros:

```
> class(vectorEdades0)
[1] integer
```

En cualquier caso, ahora el vector `vectorEdades` contiene la información del fichero `csv` con el que hemos empezado. En la Sección 4 vamos a empezar a hacer Estadística Descriptiva. Pero antes, te pedimos un poco más de paciencia, porque vamos a entretenernos un poco más en los detalles del manejo de ficheros `csv` con R. No pretendemos ser exhaustivos, ni mucho menos, pero sí queremos aprender lo suficiente como para evitarnos los problemas más habituales en los casos más sencillos. Antes de seguir adelante:

Ejercicio 3.
Guarda en un vector llamado `vectorEdades` *el contenido del fichero Tut02-Edades.csv* *Solución en la página 48.* □

Ejercicio 4.
La tabla Tut02-tabla1.csv contiene parte del famoso estudio sobre enfermedades cardiacas realizado en Framingham (USA). Hay algunos datos de las variables `sex1` *(género),* `age1` *(edad) y* `systbp1` *(presión sanguínea sistólica). Abre el fichero con el bloc de notas para echar un vistazo; fíjate en los separadores que s eutilizan y en que cada columna tiene un nombre. Utiliza el comando*

```
tabla = read.table(file = "../datos/Tut02-tabla1.csv", sep =";", header = TRUE)
```

para leer la tabla. Fíjate que `sep` *indica qué separador se ha usado para distinguir unos campos de otros y* `header=TRUE` *pide a R que use la primera fila de la tabla para los nombres de las columnas. Se pide:*

1. *Guarda en vectores diferentes cada columna de la tabla.*
2. *Averigua de qué clase es cada variable (vector).*

Solución en la página 48. □

Como veremos en seguida, no es necesario memorizar las opciones de cada comando (como `sep` o `header`). Para cerrar este apartado, es conveniente haber visto el error que se produce cuando el fichero que tratamos de leer desde R no existe, o no está en el directorio de trabajo.

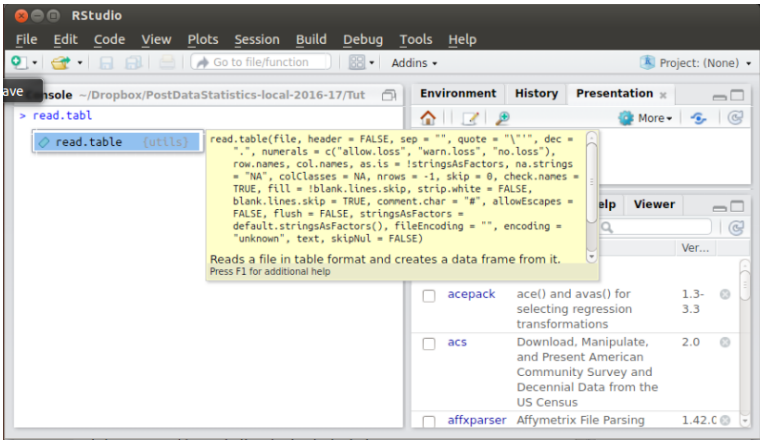
Ejercicio 5.
Prueba a ejecutar:

```
errorDeLectura = read.table(file = "../datos/EsteFicheroNoExiste.csv")
```

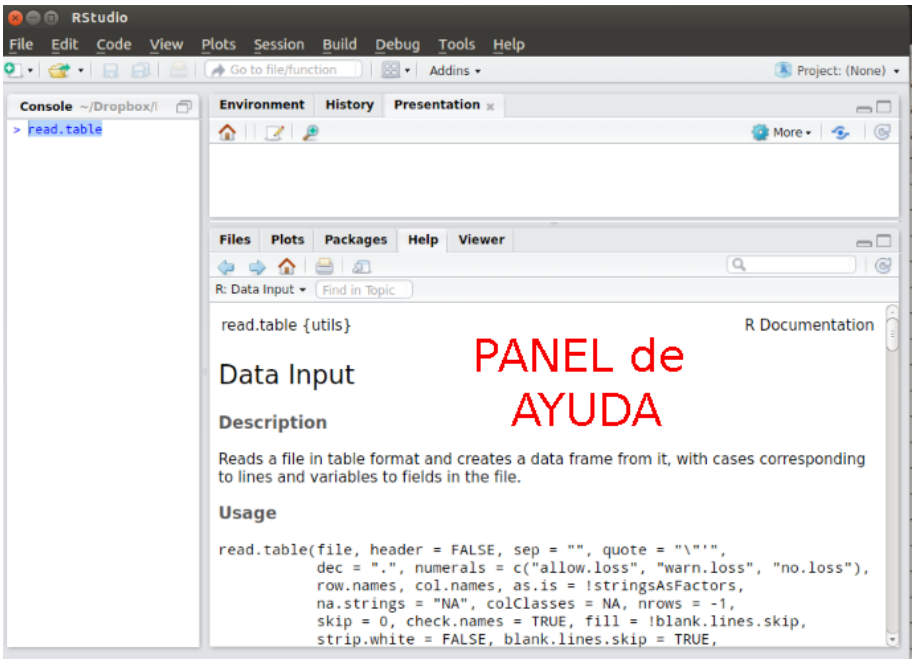
y fíjate en el mensaje de error. Solución en la página 48. □

3.1. Cómo obtener ayuda y usar el tabulador en RStudio.

Vamos a aprovechar estos ejemplos de lectura de ficheros `csv` para aprender a obtener ayuda sobre las funciones de R. La mayoría de los comandos de R, como el comando `read.table` que hemos usado aquí, tienen muchos argumentos opcionales. Y hay muchísimos comandos en R (cada vez más, porque, como veremos más adelante, se pueden añadir comandos nuevos a R cuando se necesitan). No es fácil recordar todos esos comandos, cada uno con sus opciones, así que es muy importante aprender a obtener esa información cuando la necesitamos.
Para ver cómo obtener esa función en RStudio, sitúate en una línea vacía de la Consola de Comandos, escribe `read.tabl` (ojo, que no hay que escribir el nombre completo) y pulsa la tecla del tabulador. Verás aparecer una ventana emergente con información sobre la función `read.table`, como en esta figura:



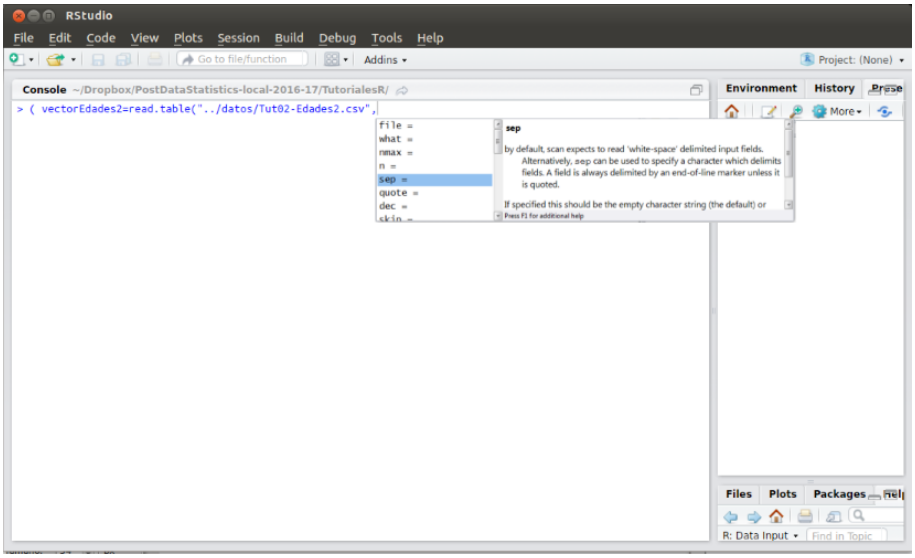
Como ves, se completa el nombre de la función y muestra un resumen de la función `read.table`, que a veces será suficiente. Si necesitas más detalles, haz caso a la indicación que aparece al final de esa ventana, y pulsa la tecla `F1`. Al hacerlo, verás que en el panel inferior derecho de RStudio aparece el fichero de ayuda de la función `read.table`, como en esta figura:



Esos ficheros de ayuda son muy útiles, aunque al principio pueden resultar un poco intimidatorios, ya sea por un exceso de información, ya sea porque se da por supuesto que el usuario ya entiende toda una serie de conceptos de R. Podemos garantizar al lector que, a medida que se progresa en R, la utilidad de estos ficheros aumenta.
Pero la utilidad de la tecla tabulador en RStudio no se acaba ahí. Supongamos que hemos empezado a usar la función `read.table`, como antes, para cargar el contenido del fichero `Tut02-Edades2.csv` en un vector. Pero se nos ha olvidado cual era el argumento que nos permitía elegir la coma como separador. No pasa nada, cuando tenemos escrita una parte del comando, como en

```
( vectorEdades2=read.table("../datos/Tut02-Edades2.csv",
```

entonces, después de teclear la coma (sin la coma no funcionará), pulsamos el tabulador y una nueva ventana emergente nos informa de los posibles argumentos que (probablemente) podemos querer usar.

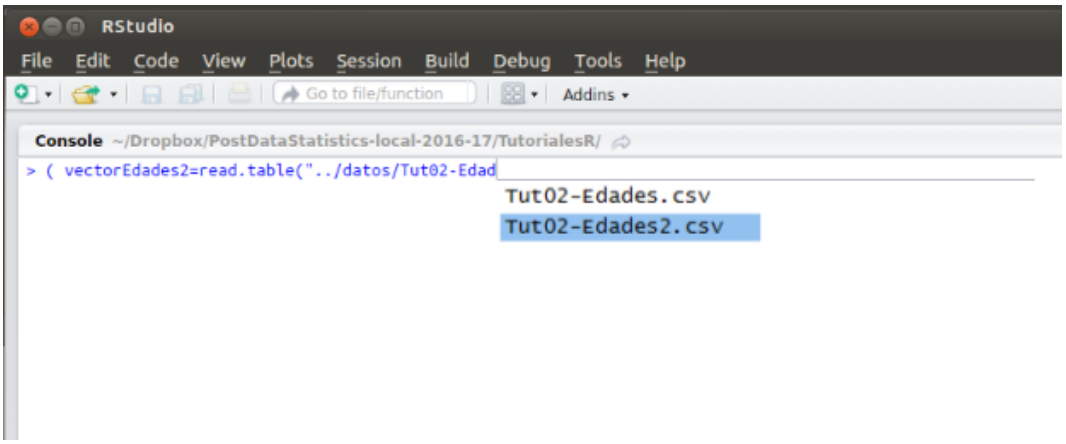


Entre ellos (en quinto lugar) aparece `sep`, y al colocar el ratón sobre él, obtenemos de nuevo una explicación más detallada.

Finalmente, para terminar de intentar venderte la idea de que, en RStudio, el tabulador es tu amigo, imagínate que, de nuevo, quieres aplicar `read.table` para leer el fichero `Tut02-Edades2.csv`. Pero te da pereza escribir un nombre tan largo, o no recuerdas exactamente el nombre, o dudas entre dos nombres... no importa cual sea el motivo, empiezas a escribir el nombre, por ejemplos las dos primeras letras, como en

```
( vectorEdades2 = read.table("Tu
```

y, de nuevo, pulsas el tabulador. Entonces RStudio te ofrece, en una nueva ventana emergente, todas las posibles formas que se le ocurren de completar el nombre, teniendo en cuenta los objetos que hayas definido hasta ese momento. En nuestro ejemplo, el resultado es el de esta figura:



Y ahora puedes seleccionar, entre esas dos opciones, la que quieras utilizar. En general, RStudio nos ofrece bastante ayuda, para hacer más cómodo nuestro trabajo con R, y el tabulador es una de las herramientas más versátiles y útiles.

3.2. El camino contrario: escribir un vector en un fichero de tipo `csv`.

Hemos aprendido a leer con R tablas de datos a partir de ficheros de tipo `csv` y guardar en una vector el contenido de una de sus columna. A partir de ahí, podremos trabajar con esos datos. Muchas veces, después de hacer operaciones en R, obtendremos como resultados nuevos vectores (y, más adelante en el curso, otro tipo de objetos). Lo natural, entonces, es aprender a guardar ese vector en un fichero de tipo `csv`, como el fichero con el que empezamos. De esa forma, por ejemplo, puedes compartir tus resultados con otras personas que utilicen programas distintos de R. Para ello, si antes usábamos la función `read.table`, ahora vamos a usar la función `write.table` de R.

Ejercicio 6. Copia y pega *los siguientes datos para crear un vector de R llamado* `muestra`.
29, 17, 63, 31, 55, 9, 92, 61, 10, 16, 63, 6, 61, 59, 66, 41, 68, 6, 99, 21, 87, 68, 52, 83, 66, 98, 45, 50, 24, 100, 83, 37, 44, 4, 97, 67, 56, 74, 75, 71, 55, 22, 86, 22, 93, 65, 38, 84, 54, 83, 100, 71, 99, 19, 63, 11, 11, 62, 91, 20, 79, 42, 59, 95, 70, 74, 8, 25, 45, 58, 57, 75, 81, 34, 70, 68, 39, 12, 14, 21 □

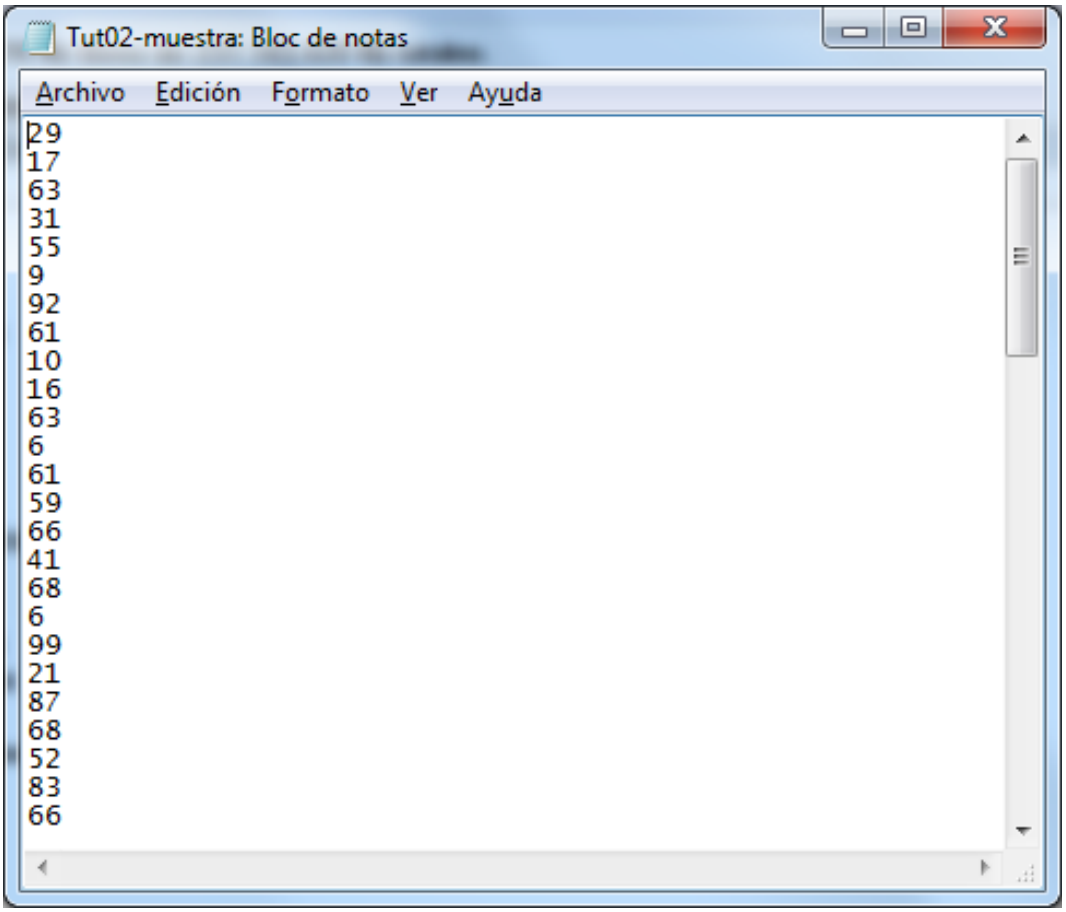
Para guardar ese vector `muestra` en un fichero `csv` (dentro de la subcarpeta `datos` del *directorio de trabajo*) podemos utilizar la función `write.table`, usando como argumentos:

- El nombre del vector que queremos guardar, que es `muestra`.
- Y el nombre del fichero en el que lo guardamos. Vamos llamarlo, por ejemplo, `Tut02-muestra.csv`. Le indicamos a R, además, que lo ponga en la subcarpeta `datos`.
- Además, vamos a guardar los datos en una columna de una tabla. Las tablas pueden tener nombre en las columnas y en las filas. Imagina una tabla con datos de pacientes: cada fila corresponde a un paciente (y lleva un identificador) y en cada columna se guardan los valores de una variable para todos los pacientes (edad, género,...). Por eso es preciso gestionar los nombres de fila y columna (en inglés, row y column, respectivamente).

Seguro que ya entiendes los argumentos que vamos a emplear en la función `write.table`

```
write.table(muestra,file = "../datos/Tut02-muestra.csv",
            sep = ";", row.names = FALSE, col.names = FALSE)
```

Si abres el fichro `csv` resultante con el bloc de notas verás lo siguiente



Puedes comprobar qué sucede si no usas alguno de los argumentos. Por ejemplo, prueba con

```
write.table(muestra,file = "../datos/Tut02-muestra.csv")
```

o bien ejecuta

```
write.table(muestra,file = "../datos/Tut02-muestra.csv",
            sep = ";", row.names = FALSE)
```

o cualquier otra combinación que se te ocurra e inspecciona los ficheros resultantes con el bloc de notas para comprobar el resultado.

4. Estadística descriptiva de una variable cuantitativa, con datos no agrupados.

Para seguir adelante, tenemos que empezar por aprender un poco más sobre la forma en la que R permite operar con los vectores. Vamos a hacerlo a través de una serie de ejercicios-ejemplos.

Ejercicio 7.

1. *Vamos a definir dos vectores, cada uno de ellos con* 12 *números enteros. El primero, al que llamaremos* `vector1`*, está formado por:*

   8, 5, 19, 9, 17, 2, 28, 18, 3, 4, 19, 1.

   Usa copiar y pegar *a partir de este fichero pdf para definir el* `vector1` *en R.*

2. *El segundo vector, al que en un rapto de originalidad llamaremos* `vector2`*, está almacenado en el fichero:*

   Tut02-vector-2.csv

   Guárdalo en `datos` *y examínalo primero con un editor de texto (como el* Bloc de Notas*), para ver cómo están almacenados los datos en ese fichero. Una vez hecho esto, usa la función* `read.table` *para almacenar el contenido de ese fichero csv en el* `vector2` *de R.*

Solución en la página 48. □

4.1. Aritmética vectorial en R.

Ahora vamos a sumar los dos vectores. Ejecuta el comando:

```
vector1 + vector2
```

para comprobar que el resultado es la suma, componente a componente, de los dos vectores:

```
> vector1 + vector2
 [1] 26 23 39 21 37  3 58 24 29 24 34 26
```

Este resultado es bastante previsible. Lo que quizá resulté un poco más inesperado es que si multiplicas los dos vectores, también se obtienen los productos componente a componente: el primer elemento de `vector1` se multiplica por el primero de `vector2`, el segundo por el segundo, etc.:

```
> vector1 * vector2
 [1] 144  90 380 108 340   2 840 108  78  80 285  25
```

Esta forma de operar de R, como veremos pronto, es muy útil. Si lo piensas un momento, cada vector numérico de R se corresponde con una columna de una hoja de cálculo, como las que usábamos en el Tutorial01. Allí vimos que, para algunas operaciones, era necesario multiplicar dos columnas, elemento a elemento. En Calc esa operación no es complicada, pero en R es aún más fácil. Basta con un comando como `vector1 * vector2` para conseguir el objetivo. Veremos muchos más ejemplos, a lo largo del curso, en los que una operación que, en Calc, requeriría bastantes pasos, en R se reduce a ejecutar un único comando.
En R, las operaciones con vectores siguen, en general, este esquema de funcionamiento, componente a componente. Si, por ejemplo, ejecutas

```
> vector1 + 5
 [1] 13 10 24 14 22  7 33 23  8  9 24  6
> 2 * vector1
 [1] 16 10 38 18 34  4 56 36  6  8 38  2
```

verás que, en el primer caso, cada uno de los elementos de `vector1` se ha incrementado en 5, mientras que, en el segundo caso, cada uno se ha multiplicado por 2. Sucede lo mismo con otras operaciones, como elevar al cuadrado:

```
> vector1^2
 [1]  64  25 361  81 289   4 784 324   9  16 361   1
```

Y podemos aplicar funciones en un solo paso. Por ejemplo, podemos calcular el logaritmo neperiano de todos los elementos de un vector en un sólo paso:

```
> log(vector1)
 [1] 2.0794415 1.6094379 2.9444390 2.1972246 2.8332133 0.6931472
 [7] 3.3322045 2.8903718 1.0986123 1.3862944 2.9444390 0.0000000
```

Los resultados de todas estas operaciones son, de nuevo, vectores. Y, por tanto, se pueden asignar a nuevas variables para utilizarlos en nuestras operaciones. Por ejemplo, si luego vas a necesitar los logaritmos de `vector1`, puedes definir una variable para identificar ese vector, llamada `logVector1` (e el nombre que tú quieras), haciendo:

```
> logVector1 = log(vector1)
```

Recuerda que si, además de la asignación, quieres ver el resultado, puedes rodear toda esta sentencia con paréntesis. De lo contrario R la ejecuta, pero no imprime ninguna respuesta, como ya sabes.

Aunque al principio puede parecer complicado, cuando te acostumbres a esta forma de trabajar con vectores, verás que resulta muy cómoda y eficiente.

Generando vectores de números enteros consecutivos

Hay muchas ocasiones en que, para alguna operación, necesitamos un vector que contenga los números enteros consecutivos del 1 al n, o del 0 al n, etc. En R esto es muy fácil de hacer. Por ejemplo, para conseguir un vector con los números del 1 al 15 basta con ejecutar:

```
> 1:15
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
```

Esa notación con los dos puntos permite generar vectores en los que cada elemento es igual al anterior más 1. Otro ejemplo:

```
> -7:13
 [1] -7 -6 -5 -4 -3 -2 -1  0  1  2  3  4  5  6  7  8  9 10 11 12 13
```

O incluso, cuando los puntos de partida no son enteros:

```
> 1.5:7.5
[1] 1.5 2.5 3.5 4.5 5.5 6.5 7.5
> 1.5:7.4
[1] 1.5 2.5 3.5 4.5 5.5 6.5
```

¿Ves la diferencia entre estos dos casos?

Es muy posible que te estés preguntando ¿y si quiero ir avanzando de 2 en 2, o quiero ir hacia atrás ("avanzando" de -2 en -2, por ejemplo)? Pronto, cuando hayamos aprendido un poco más, volveremos sobre esto, y comprobaremos que R nos ofrece un catálogo de herramientas extremadamente potentes, para fabricar vectores de acuerdo con nuestros deseos y necesidades.
Los vectores fabricados con este método son vectores iguales que los demás, y se pueden asignar a variables, operar con ellos, etc.

Ejercicio 8.

1. *Fabrica un vector con los cuadrados de los números del* 1 *al* 10*, y asígnalo a la variable* `cuadrados`*. Usa los paréntesis para ver el resultado de la asignación.*
2. *Haz lo mismo con los cuadrados de los números del* 2 *al* 11*, y un vector* `cuadrados2`*.*
3. *Resta los vectores* `cuadrados2 - cuadrados`*. ¿Observas algo interesante?*

Solución en la página 48. □

Funciones vectoriales

Hemos visto antes que se puede aplicar una función, como la raíz cuadrada, a un vector, y se obtiene como resultado otro vector con las raíces cuadradas, elemento a elemento. Pero hay otro tipo de funciones que permiten hacer cálculos con el vector *en conjunto*. Por ejemplo, antes hemos fabricado el vector `-7:13`, y la función `length` nos dice cuantos elementos tiene ese vector:

```
> length(-7:13)
[1] 21
```

Puedes obtener la longitud del vector con `length(vectorEdades)`. Para comprobarlo, suponiendo que el fichero `Tut02-Edades.csv` está en la subcarpeta `datos` del *directorio de trabajo*, prueba a ejecutar:

```
> df.Edades = read.table(file="../datos/Tut02-Edades.csv")
vectorEdades <- df.Edades$V1
> length(vectorEdades)
[1] 100
```

Como ves, el resultado de `length` es el esperado.

Vamos a conocer otra función, llamada `sum`, que sirve para calcular la suma de todos los elementos de un vector:

```
> sum(vectorEdades)
[1] 675
```

Ejercicio 9.
¿Cuánto vale la suma de los cuadrados de los números del 1 *al* 100*?* *Solución en la página 48.* □

A lo mejor te ha pasado desapercibido, pero con las funciones `length` y `sum` estamos, por fin, en condiciones de empezar a hacer Estadística Descriptiva. Por ejemplo, podemos calcular la media aritmética de `vectorEdades`:

```
> ( mediaEdades = sum(vectorEdades) / length(vectorEdades) )
[1] 6.75
```

Y una vez que tenemos la media, podemos calcular la varianza poblacional. Para que resulte más claro, vamos a mostrarte el proceso paso a paso, aunque al final lo ejecutaremos en un sólo paso:

1. Empezamos restando la media, que está en la variable `mediaEdades`, de cada uno de los elementos de `vectorEdades`:

   ```
   vectorEdades - mediaEdades
   ```

   El resultado es un *vector de diferencias*.

2. Ahora elevamos cada una de esas diferencias al cuadrado:

   ```
   ( vectorEdades - mediaEdades )^2
   ```

   El resultado de este paso es, de nuevo, un *vector*, ahora con las diferencias al cuadrado. Fíjate en el paréntesis, para garantizar que elevamos al cuadrado la diferencia. Si no incluyéramos el paréntesis ¿qué calcularíamos?

3. A continuación sumamos todas esas diferencias al cuadrado, usando la función `sum`:

   ```
   sum( ( vectorEdades - mediaEdades )^2)
   ```

   De nuevo, el paréntesis es fundamental, para que el cuadrado del vector de diferencias se calcule antes que la suma. En este paso, el resultado ya no es un vector, sino un *número*.

4. Por último, dividimos esa suma por n, el número de elementos de `vectorEdades`, que se obtiene con la función `length`:

   ```
   sum( ( vectorEdades - mediaEdades )^2) / length(vectorEdades)
   ```

El resultado de estos pasos se puede condensar en un sólo comando de R, en el que a la vez calculamos la varianza y la asignamos a una variable:

```
> ( varEdades = sum( ( vectorEdades - mediaEdades )^2) / length(vectorEdades) )
[1] 1.9875
```

Ejercicio 10.

1. *Calcula la desviación típica (poblacional) de* `vectorEdades`.
2. *Calcula la cuasivarianza y cuasidesviación típica (muestrales) de* `vectorEdades`.

3. *¿Qué sucede si en el tercer paso ejecutas el siguiente comando?*
   `sum( vectorEdades - mediaEdades )^2`
   ¿Por qué sucede esto?

Solución en la página 49. □

Con estos cálculos hemos empezado a hacer Estadística con R. Naturalmente, nos queda muchísimo que aprender sobre los vectores de R. Por ejemplo, como se accede a una componente individual del vector, para responder a preguntas como *"¿Cuánto vale el quinto elemento del vector?"* Más en general, aprenderemos a extraer aquellos elementos del vector que cumplan alguna condición, respondiendo a preguntas como *"¿Qué elementos del vector son mayores que* 7*?"*

4.2. Y, por fin, hagamos Estadística Descriptiva.

En esta sección vamos a trabajar con un vector de datos contenido en este fichero adjunto:

Tut02-var3.csv

Este vector contiene los datos de la variable `var3` del fichero `Tut01-PracticaConCalc.csv`, que ya hemos utilizado en el Tutorial-00 para aprender a hacer Estadística Descriptiva con Calc. En esta sección vamos a reproducir los resultados que obtuvimos allí, pero esta vez usando R. Para preparar el terreno vamos a hacer limpieza.

Ejercicio 11.

1. *Cierra Rstudio y vuelve a abrirlo, para limpiar la memora del sistema. El trabajo que hemos hecho hasta ahora se perderá (aunque los comandos siguen en el Historial de Comandos que ya vimos).*
2. *Establece el* Directorio de Trabajo *(por ejemplo, el* Escritorio*), y asegúrate de que* `Tut02-var3.csv` *está guardado en la subcarpeta* `datos` *de ese directorio.*
3. *Carga de nuevo el contenido de ese fichero en un vector* `var3` *de R.*

Solución en la página 49. □

Verás, en lo que sigue, que siempre que calculamos un objeto le asignamos una variable. Eso permite ahorrarse la repetición innecesaria de cálculos, si más adelante volvemos a necesitar ese objeto. Y en algunos casos, como veremos más adelante, al hacer esto podemos acceder a propiedades del objeto calculado que R no muestra en su respuesta por defecto.

Rango, máximo y mínimo.

Para empezar, vamos a aprender tres nuevas funciones de R que nos permiten calcular esos tres valores:

```
> (minimo = min(var3) )
[1] 0
> (maximo = max(var3) )
[1] 16
> (rango = range(var3) )
[1]  0 16
```

El funcionamiento de las dos primeras es evidente. La función `range` devuelve un *vector*, formado por el máximo y el mínimo.

Tablas de frecuencia. Clases de objetos en R

La tabla de frecuencia absoluta de `var3` se obtiene usando la función `table`, de esta forma tan sencilla:

```
> ( tablaFrecAbs = table(var3) )
var3
  0   1   2   3   4   5   6   7   8   9  10  11  12  13  14  16
  9  25 100 188 244 246 186 131  87  43  28   6   2   2   2   1
```

La primera fila contiene los elementos de `var3`, y en la segunda aparece la frecuencia de cada uno de ellos. Puedes comprobar que son los mismos resultados que obtuvimos, con Calc, en la página **??** del Tutorial01. Aunque allí, desde luego, el proceso fue más laborioso.

A lo mejor te has dado cuenta de que el formato de la respuesta es diferente de los que hemos obtenido hasta ahora. Para empezar, la respuesta no empieza con el `[1]` al que ya nos vamos acostumbrando. El motivo de esto es que el objeto que hemos obtenido como respuesta no es un vector, sino una tabla. Ya dijimos que R utiliza distintos tipos de objetos, con fines distintos. ¿Cómo podemos saber de qué clase es un objeto? Pues usando una función de R, oportunamente llamada `class`. Si la usamos con el vector `var3` y con su tabla de frecuencias, obtenemos:

```
> class(var3)
[1] "numeric"
> class(tablaFrecAbs)
[1] "table"
```

El vector `var3` es de clase `numeric` porque es un vector de números. Y la tabla de frecuencias es de clase `table`. Veremos a lo largo del curso otros objetos de tipo *tabular*, como las matrices (clase `matrix`), los conjuntos de datos (clase `data.frame`) y los `arrays`.
Aunque las tablas son objetos distintos de los vectores, tienen algunas propiedades en común, y algunas de las funciones que hemos usado con vectores también se pueden usar con tablas.

Ejercicio 12.
Aplica las funciones `length`*,* `sum`*,* `min`*,* `max` *y* `range` *a la tabla* `tablaFrecAbs`*. Solución en la página 49.* □

Siguiendo con los parecidos entre tablas y vectores, puedes multiplicar (o dividir) una tabla por un número, y la operación se hará elemento a elemento:

```
> 2 * tablaFrecAbs
var3
  0   1   2   3   4   5   6   7   8   9  10  11  12  13  14  16
 18  50 200 376 488 492 372 262 174  86  56  12   4   4   4   2
```

Como ves, cada frecuencia se multiplica por 2. Esta propiedad de las tablas permite obtener, de forma, muy sencilla la tabla de frecuencias relativa de `var3`. Como primer paso, vamos a guardar la longitud de `var3` en la variable `n`:

```
> ( n = length( var3 ) )
[1] 1300
```

Y ahora podemos dividir la tabla por `n` para conseguir la tabla de frecuencias relativas (hemos estrechado la *Consola de Comandos* para que el resultado cupiera en esta página).

```
> ( tablaFrecRel = tablaFrecAbs / n )
var3
           0            1            2            3            4
0.0069230769 0.0192307692 0.0769230769 0.1446153846 0.1876923077
           5            6            7            8            9
0.1892307692 0.1430769231 0.1007692308 0.0669230769 0.0330769231
          10           11           12           13           14
0.0215384615 0.0046153846 0.0015384615 0.0015384615 0.0015384615
          16
0.0007692308
```

Ejercicio 13.
Ya sabemos cuanto debe sumar esta tabla de frecuencias relativas. Compruébalo. Solución en la página 49. □

Si lo que queremos es una tabla de frecuencias acumuladas, debemos emplear la función `cumsum`.

```
> ( tablaFrecAcu = cumsum(tablaFrecAbs)  )
   0    1    2    3    4    5    6    7    8    9   10   11   12   13   14   16
   9   34  134  322  566  812  998 1129 1216 1259 1287 1293 1295 1297 1299 1300
```

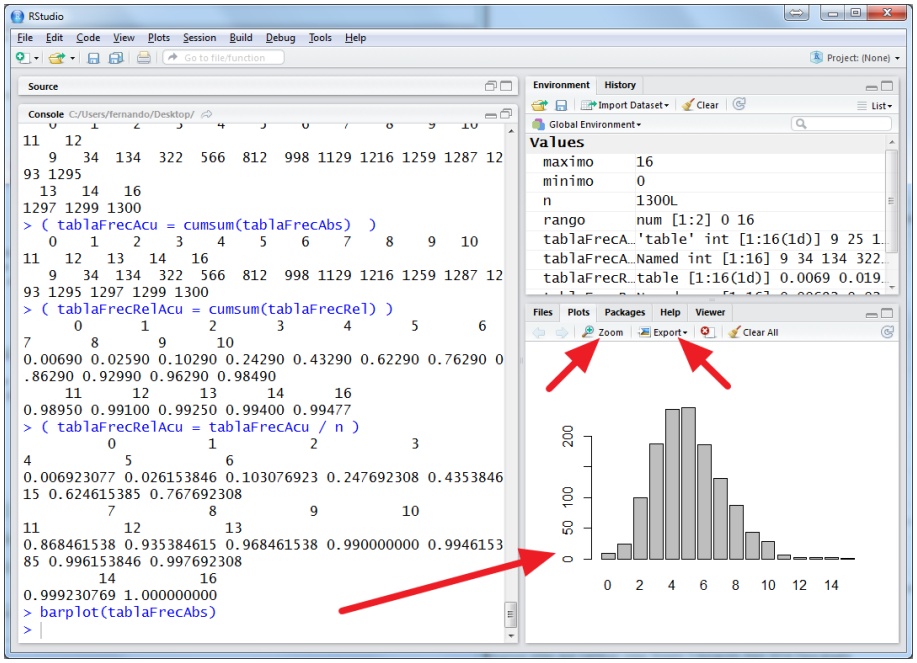
Ejercicio 14.
La última tabla que falta es la de frecuencias relativas acumuladas, pero ya tienes todos los ingredientes para obtenerla, de dos maneras diferentes. Hazlo, y comprueba si dan el mismo resultado. Si no es así, ¿a qué crees que se debe, y cuál es la mejor opción? Asigna el resultado a la variable `tablaFrecRelAcu`. *Solución en la página 50.* □

Gráficos de columnas y diagramas de caja (boxplot). Argumentos con nombre.

Vamos a fabricar nuestros primeros gráficos con R. En concreto, vamos a dibujar un gráfico de barras a partir de la tabla de frecuencias de `var3`, usando la función `barplot` de R:

```
> barplot(tablaFrecAbs)
```

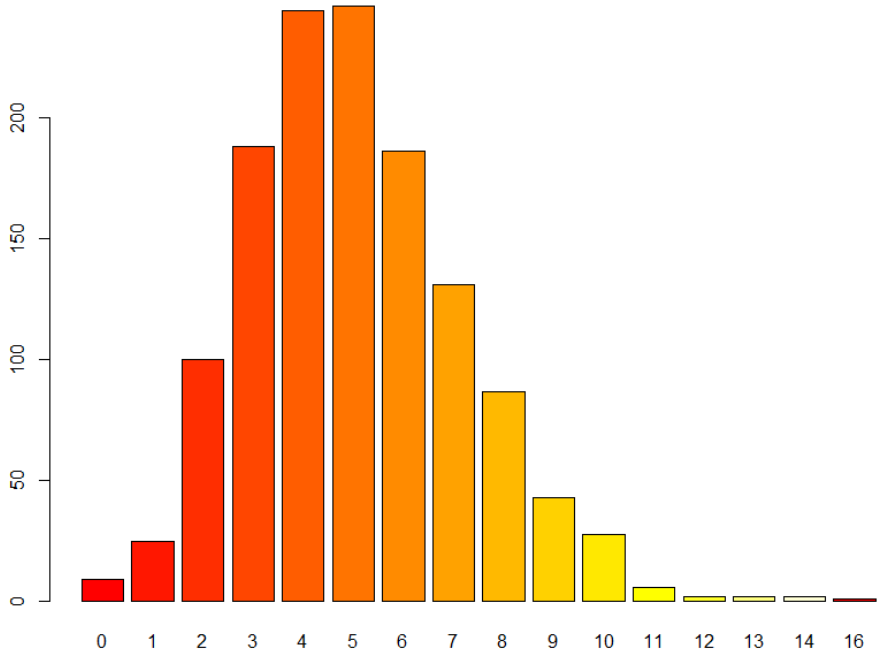
Al ejecutar este comando no debes ver ninguna salida en la `Consola de Comandos`. En su lugar, en el panel inferior derecho de RStudio, concretamente en la ventana *Plots*, aparecerá el gráfico deseado. El resultado puede variar, dependiendo de la anchura de tu ventana de RStudio.



Hemos destacado además, en la figura, los dos botones *Zoom* y *Export* que te permiten, respectivamente, ver una versión ampliada de la Figura (que se redimensiona dinámicamente si cambias la ventana), y guardar esa figura como un fichero de gráficos que puedes insertar en cualquier documento, editar con un programa de manipulación de gráficos, etcétera.
Los gráficos de R se pueden controlar hasta en sus detalles más mínimos. Como aperitivo mínimo, vamos a añadir algo de color al gráfico:

```
>  barplot(tablaFrecAbs, col = heat.colors(15))
```

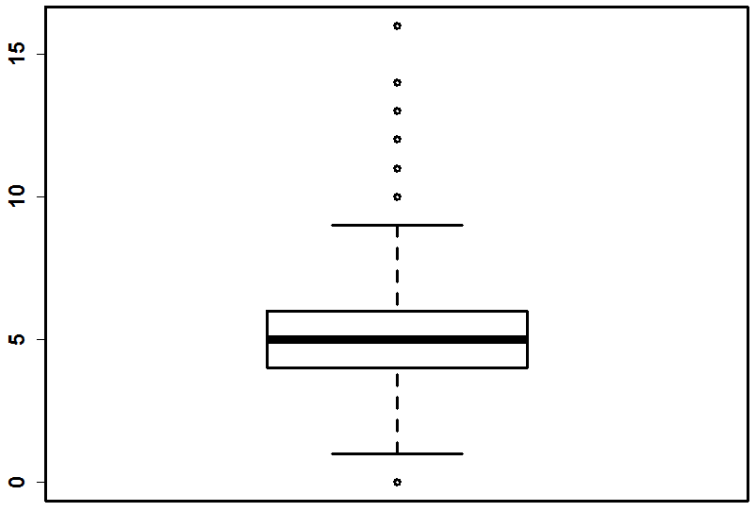
El resultado es esta figura:



Las posibilidades de control de los gráficos de R son, como decimos, inmensas y pueden llegar a ser abrumadoras. Hay libros, de cientos de páginas, dedicados a este tema, como el *R Graphics Cookbook* de W. Chang (ver las referencias del curso). Nosotros, a lo largo de los tutoriales del curso, vamos a aprender algo más, aunque a penas nos podremos limitar a rozar la superficie de lo que se puede llegar a hacer.
En la Sección 2.3.1 del libro hemos descrito los diagramas de cajas, o boxplots (recuerda la Figura 2.2, de la pág. 36 del libro). Y en el Tutorial-01 dijimos que con Calc no es fácil dibujar ese tipo de diagramas. En cambio, con R, no podría ser más fácil:

```
> boxplot(var3)
```

Y el resultado es esta figura:



En realidad tenemos que confesar que el resultado es ligeramente distinto, y que hemos manipulado algunos de los parámetros gráficos de R, para que las líneas fueran más gruesas, y figura resultara más fácil de visualizar. Ya aprenderemos esos trucos. Por el momento, nos limitamos a insistir en lo fácil que resulta obtener el diagrama en R.
Si deseas que la orientación del diagrama sea horizontal, basta con ejecutar

```
boxplot(var3, horizontal=TRUE)
```

y también puedes controlar si se incluyen, o no, los valores atípicos (*outliers*), con el argumento `outline`. Para excluirlos, ejecutaríamos

```
boxplot(var3, outline = FALSE)
```

Una advertencia: en este, como en otros casos, aunque lo estamos presentando por separado, siempre puedes combinar a la vez varios argumentos opcionales. Por ejemplo, no hay problema en ejecutar:

```
boxplot(var3, horizontal = TRUE, outline = FALSE)
```

El único problema que puedes tener es el *orden* en que colocas esos argumentos opcionales. Por ejemplo, si ejecutas:

```
boxplot(var3, TRUE, FALSE)
```

entonces R tendrá que decidir por su cuenta (aplicando las reglas que, en su momento, decidieron los programadores), a cuál de sus argumentos opcionales corresponde esos valores `TRUE` y `FALSE`. ¡Y esos argumentos opcionales son muchos, así que el resultado puede ser muy distinto de lo que esperábamos!

Ejercicio 15.
Ejecuta ese comando para ver el gráfico resultante. *Solución en la página 50.* □

Para evitar ese problema, R utiliza los *argumentos con nombre*, como en `horizontal = TRUE`. Aquí, `horizontal` es el nombre del argumento, y `TRUE` es el valor que le hemos asignado al llamar a la función `boxplot`. Gracias a este mecanismo, podemos conseguir que el orden de los argumentos sea irrelevante.

Ejercicio 16.
Ejecuta de nuevo el comando:

```
boxplot(var3, horizontal = TRUE, outline = FALSE)
```

Y ahora, con los argumentos opcionales intercambiados de posición:

```
boxplot(var3, outline = FALSE, horizontal = TRUE)
```

Verás que no hay ninguna diferencia. *Solución en la página 50.* □

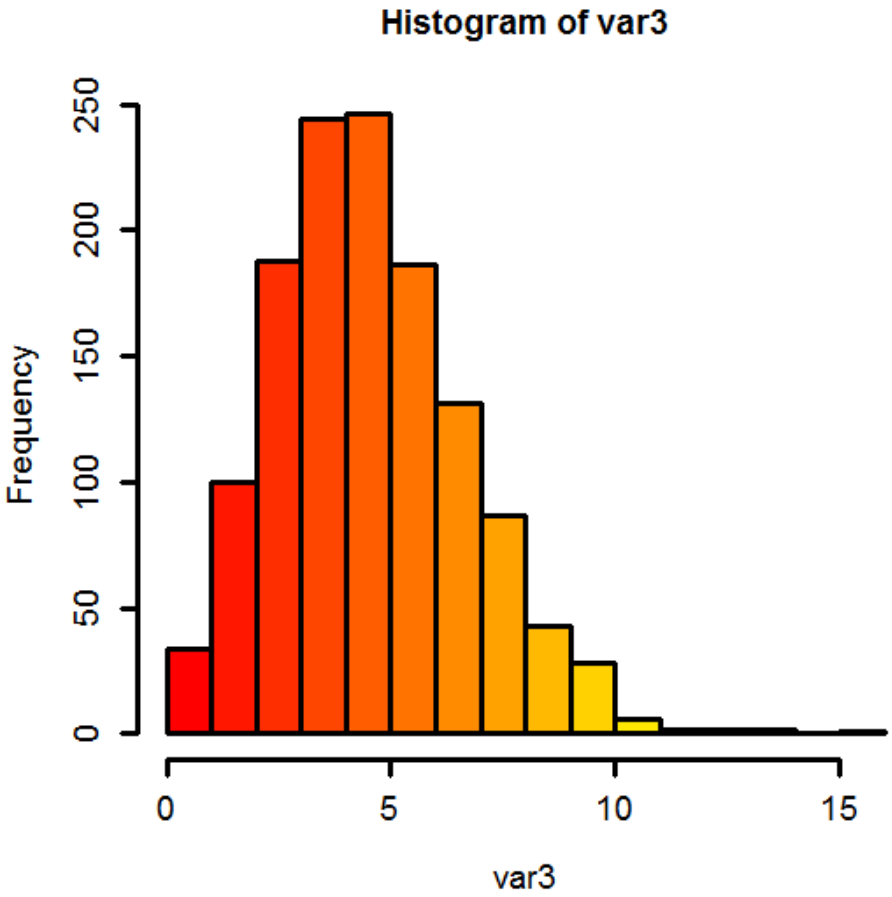
Para completar, por el momento, la discusión sobre gráficos elementales, queremos comentar que también se pueden dibujar diagramas de sectores en R. Pero vamos a ser consecuentes con nuestra recomendación de no utilizar ese tipo de gráficos, y no incluiremos instrucciones. Siempre puedes buscar en Internet, si realmente quieres dibujarlos.

Histogramas en R

En el Tutorial-01 dijimos que no es fácil usar Calc para dibujar histogramas, y nos comprometimos a mostrar la forma de dibujar ese tipo de gráficos con R. Todavía no estamos en condiciones de aprovecharlos al máximo, porque no hemos visto las herramientas que R nos proporciona para agrupar una variable en clases, y para trabajar después con esas clases. Veremos todo eso más adelante, pero para adelantar algo, vamos a usar la variable `var3`, para comprobar lo fácil que es obtener su histograma, usando el comando:

```
hist(var3, col = heat.colors(15))
```

El argumento `, col = heat.colors(15)` es opcional, y sólo sirve para fijar la misma combinación de colores que ya usamos en el diagrama de barras. Como en otros casos, hemos manipulado ligeramente el comando para que las líneas sean más gruesas y la visualización sea más fácil. Tú obtendrás un trazo más tenue; ya aprenderemos a hacer ese tipo de ajustes):



Como puedes ver, en este caso tan sencillo la información que transmiten el diagrama de barras y el histograma es, esencialmente, la misma. Fíjate en el detalle técnico de que, en el histograma, no existe la separación entre las barras que sí se daba en el otro diagrama. Veremos más adelante ejemplos en los que la diferencia ente ambos tipos de gráficos será mucho más marcada.

Media aritmética.

Al final de la Sección 4.1 hemos visto como calcular la media aritmética y la varianza del vector `vectorEdades` *"a mano"*, usando la aritmética vectorial. Pero, desde luego, esas operaciones son tan básicas, que, como sucedía en Calc, es de esperar que R incluya funciones para calcularlas en un paso.
Así, si queremos calcular la media aritmética de `var3`, basta con usar la función `mean`:

```
> (media = mean(var3))
[1] 5.039231
```

Ejercicio 17.

1. *Asegúrate de tener disponible en R el* `vectorEdades`*, y calcula su media, usando la función* `mean`*. Compárala con la que obtuvimos en la página 21.*
2. *Y a la inversa, calcula la media de* `var3` *usando aritmética vectorial, sin recurrir a* `meam`*.*

Solución en la página 51. □

Cuasivarianza, y cuasidesviación típica. Varianza y desviación típica.

Con la varianza y la desviación típica, no obstante, tenemos una pequeña dificultad. Los creadores de R diseñaron el programa para hacer Estadística, no para enseñarla. Aunque las cosas han cambiado mucho desde entonces, y ahora R se usa como herramienta de enseñanza en cientos de cursos en todo el mundo, no hay que olvidar que R es, ante todo, una herramienta profesional, y ese espíritu lo impregna todo. En particular, como veremos más adelante en el curso, en Estadística, la varianza y desviación típica poblacionales juegan un papel muy secundario, comparado con el de la

cuasivarianza y cuasidesviación típica muestrales. La razón es que el trabajo se hace, casi siempre, con muestras, claro. Las cantidades poblacionales son, a menudo, objetos teóricos, esencialmente inaccesibles.
Por esas razones, R no incluye funciones para calcular ni la varianza poblacional, ni la desviación típica poblacional. Sí se incluyen, en cambio, la función `var` para calcular la cuasivarianza muestral:

```
> (varMuestral = var(var3))
[1] 4.715165
```

y la función `sd` para calcular la cuasidesviación típica muestral (que es la raíz cuadrad de la anterior):

```
  > (desvTipMuestral = sd(var3))
[1] 2.171443
```

Ejercicio 18.
Comprueba que `sd(var3)` *es la raíz cuadrada de* `var(var3)`. *Solución en la página 51.* □

¿Y si queremos calcular las cantidades poblacionales? En tal caso, basta con tener en cuenta las definiciones, y para la varianza poblacional hacemos:

```
  > ( varPobl= ( (n-1) / n ) * varMuestral )
[1] 4.711538
```

y entonces, para la desviación típica poblacional sólo hay que calcular la raíz cuadrada:

```
> ( desvTipPobl = sqrt( varPobl ) )
[1] 2.170608
```

Medidas de posición: mediana, cuartiles y percentiles. Rango intercuartílico.

Para calcular la mediana, tenemos la función `median`:

```
> ( mediana = median(var3))
[1] 5
```

El cálculo de los cuartiles (o percentiles) en R, como en cualquier otro programa estadístico, requiere una discusión más detallada. Por un lado, si lo único que se desea es obtener la información por defecto que proporciona R, dejando que sea el programa el que decida la mejor forma de proceder, entonces las cosas son muy sencillas. Basta con usar la función `summary`, así:

```
> summary(var3)
 Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
0.000   4.000   5.000   5.039   6.000  16.000
```

Como ves, la respuesta incluye varias de las cantidades que ya habíamos calculado previamente, pero también aparecen los cuartiles primero y tercero (indicados por `1st Qu.` y `3rd Qu.`, respectivamente).
Esta función `summary` no sólo sirve para vectores, sino que se puede aplicar a muchos otros objetos de R, para obtener (lo que R considera) información básica sobre ellos.

Ejercicio 19.
¿Qué tipo de objeto produce `summary` *en su respuesta? ¿Es una tabla, un vector, otra cosa?* *Solución en la página 51.* □

Además de `summary`, en R disponemos de la función `IQR` para calcular el rango intercuartílico de un vector:

```
> IQR(var3)
[1] 2
```

Puedes comprobar que la respuesta es la diferencia entre los dos cuartiles que hemos obtenido con `summary`.

Por otra parte, en lugar de utilizar `summary`, si sólo queremos uno de los cuantiles, o si lo que queremos son algunos percentiles, podemos conseguir un control más fino de la respuesta utilizando la función `quantile`. Si la aplicamos al vector, sin más, se obtiene por defecto esta respuesta:

```
> quantile(var3)
  0%  25%  50%  75% 100%
   0    4    5    6   16
```

Pero también podemos usarla con un segundo argumento: un vector en el que aparecen los porcentajes (expresados como tantos por uno) de los percentiles que queremos calcular. Por ejemplo, para pedirle a R que calcule (con el método por defecto, es decir, el que más le gusta a R) los percentiles 5 %, 15 % y 75 % de `var3`, ejecutamos este comando:

```
> quantile(var3, c(0.05,0.15,0.75))
 5% 15% 75%
  2   3   6
```

Aquí, como queremos mantener la discusión lo más simple posible, no vamos a entrar en más detalles técnicos sobre las decisiones que R toma para calcular estos números, pero queremos prevenir al lector de que las cosas se complican enseguida: en R hay varias (concretamente, nueve) formas distintas de calcular los cuartiles.

5. Ficheros de comandos R. Comentarios.

Antes de seguir adelante, tenemos que ocuparnos de un problema que, casi con seguridad, está empezando a preocuparte. Si has hecho todo el trabajo de la sección anterior, habrá escrito muchos comandos de R, para estudiar el vector `var3`. Pero imagínate que ahora queremos estudiar otro vector, por ejemplo, el `vectorEdades` de las primeras secciones de este tutorial. ¿Tenemos que empezar de nuevo, tecleando todos esos comandos para este nuevo vector? Ya sabes que esas instrucciones están guardadas en el *Historial de Comandos* (¡incluso las erróneas!). Una opción sería irlas copiando, una por una, desde ese historial a la *Consola de Comandos*, modificarlas, y ejecutarlas. Pero no parece una gran idea, en términos de productividad.
La solución es bastante sencilla, y RStudio nos ofrece todo lo que necesitamos. Lo que vamos a hacer es guardar, en un fichero de texto plano, una lista de instrucciones de R. Esos ficheros de instrucciones tendrán la extensión

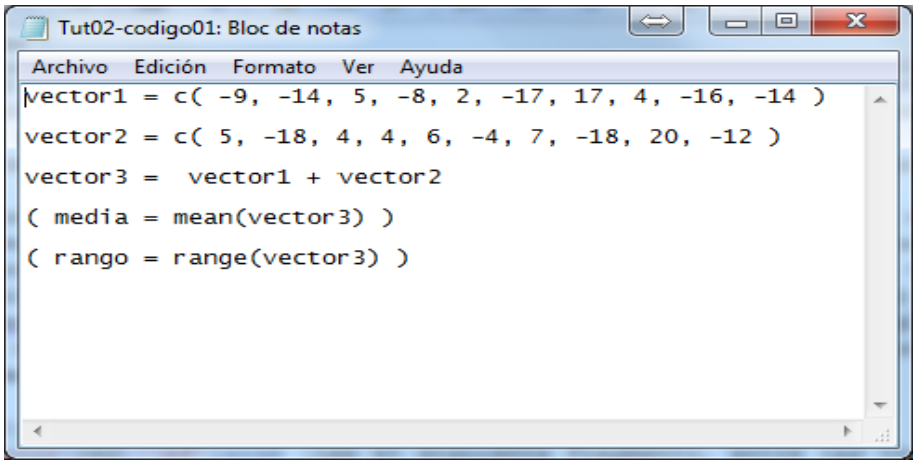
`.R`

Ya sabes que, como vimos en el Tutorial-01, la extensión sirve para identificar el tipo de fichero, y para decidir cuál es el programa predeterminado con el que el sistema abre esos ficheros (por ejemplo, al hacer doble clic sobre uno de ellos). Antes de avanzar más, veamos un ejemplo. Pero primero una advertencia, para evitar problemas: **¡Lee el siguiente fragmento, entre las líneas horizontales, antes de tratar de abrir el fichero de instrucciones!**

Cierra RStudio antes de empezar, para hacer un poco de limpieza. Hemos creado uno de esos ficheros de instrucciones, muy sencillo, que puedes abrir con un editor de texto (como el *Bloc de Notas* de Windows):

Tut02-codigo01.R,

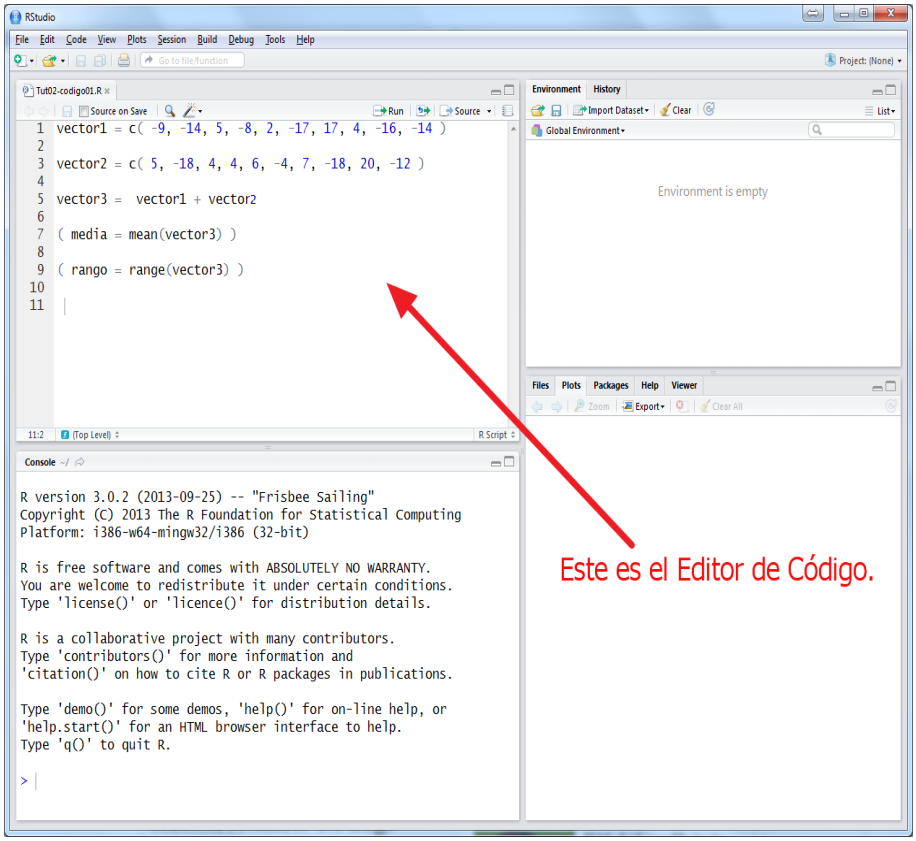
El contenido, al abrirlo con el editor es este:



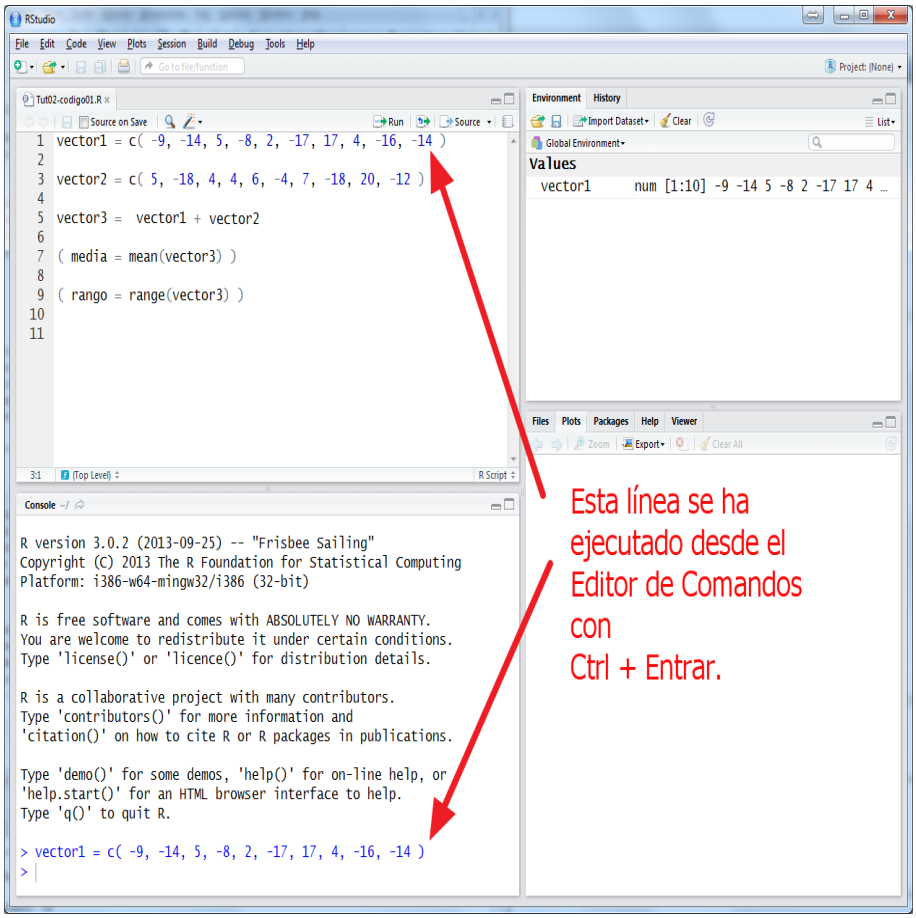
Si el ordenador en el que estás trabajando tiene instalado RStudio, como se vio en el Tutorial-00, es muy posible que el programa predeterminado para abrir los ficheros como este sea el propio RStudio. En cualquier caso, para asegurarnos, guarda este fichero en tu ordenador. ¡Recuerda usar el botón derecho para guardar! Además, es importante que lo guardes en el *Directorio de Trabajo*. **Atención:** no se trata de guardarlo en la subcarpeta `datos`, sino en el propio directorio de trabajo que contiene esa subcarpeta. Una vez guardado, usa lo que aprendimos en el Tutorial-00 para asegurarte de que, en efecto, RStudio es el programa predeterminado de tu sistema para este tipo de ficheros.

Y ahora, después de todas las maniobras anteriores, vamos a abrir el fichero con RStudio. Puedes hacer doble clic en el fichero, o si prefieres un poco más de control sobre el proceso, puedes empezar abriendo RStudio. En ese caso, usa los menús `File` → `Open File`, y en la ventana que aparece, selecciona el fichero `Tut02-codigo01.R`, y pulsa en `Abrir`.

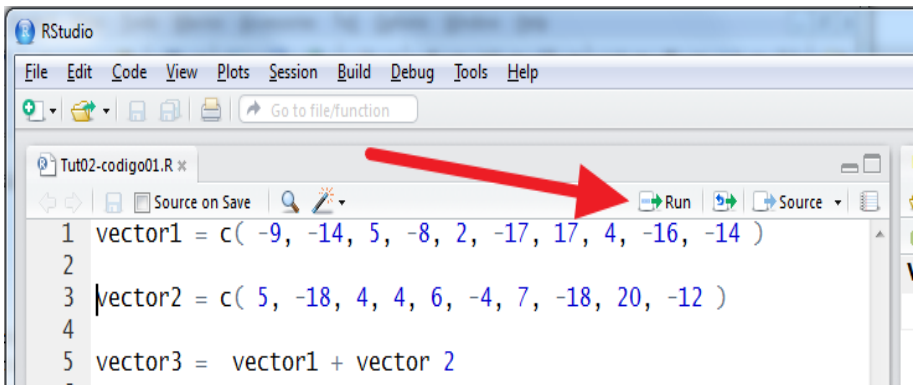
Al abrir el fichero, se abrirá una componente nueva de RStudio, que hasta ahora no habíamos utilizado. Se trata de un nuevo panel de RStudio, al que vamos a llamar el *Editor de Código* (en inglés, *Source Editor*), que se sitúa en la parte superior izquierda de la interfaz, justo encima de la *Consola de Comandos*. En ese panel aparece el contenido del fichero `Tut02-codigo01.R`,como se ve en la figura:



Este *Editor de Código* es, ante todo y como indica su nombre, un editor de texto. Es, por así decirlo, como si RStudio llevará incorporado su propio *Bloc de Notas*. Pero es, además, y como iremos viendo a lo largo del curso, un editor de texto especialmente diseñado para el trabajo con R. Habrás notado ya que, por ejemplo, se utilizan distintos colores para algunos elementos del ficheros. Además las líneas aparecen numeradas a la izquierda. Todos esos ingredientes, y muchos otros que conoceremos más adelante, están ahí para hacer nuestro trabajo con R mucho más productivo.
Para empezar, vamos a descubrir la relación que existe entre el *Editor de Código* y la *Consola de Comandos*, que hemos venido usando hasta ahora. Haz clic con el botón izquierdo del ratón en la primera línea del fichero (la que empieza con `vector1 =`), no importa en qué punto de la línea hagas clic. Asegúrate de que el cursor está situado en esa línea, y pulsa simultáneamente las teclas `Ctrl` y `Entrar`.



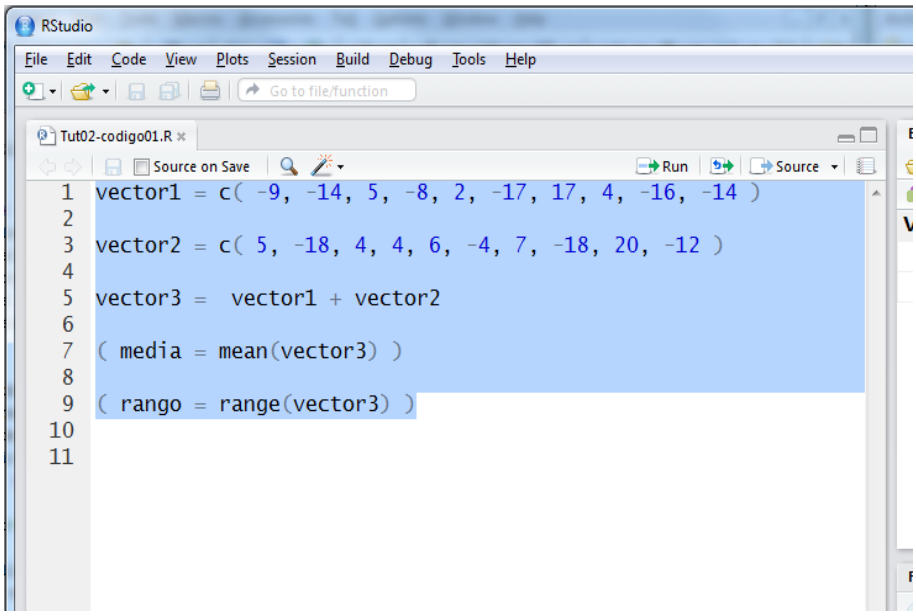
Como indica la figura, la línea de comandos en la que está situado el ratón se transfiere a la *Consola de Comandos*, y se ejecuta. Se puede conseguir el mismo resultado si, una vez situado el cursor en la línea deseada, usas el botón `Run`, situado sobre el *Editor de Código*:



Ejercicio 20.
Prueba a hacer lo mismo con el resto de las líneas de código del fichero, ejecutándolas una por una. □

Si has hecho este ejercicio, habrás visto que no hay diferencia entre ejecutar un comando desde la *Consola* o desde el *Editor*, más allá de las teclas que hay que utilizar en cada caso. A cambio, la ejecución desde el *Editor* ofrece muchas ventajas. Para empezar, puedes ejecutar todo un grupo de comandos a la vez. Para ver esto, vamos a empezar haciendo limpieza. Pulsa `Ctrl + L` (las dos teclas a la vez), para limpiar la *Consola de Comandos*. Ahora, con el ratón o el teclado, selecciona todas las líneas del *Editor de Código*. Por cierto, para moverte del *Editor de Código* a la *Consola de Comandos* y viceversa, puedes usar los atajos de teclado `Ctrl + 1` y `Ctrl + 2`.

Asegúrate de que la situación resultado es como la de esta figura (también puedes usar el atajo de teclado `Ctrl + A` para seleccionar todo el código):



Y ahora pulsa `Ctrl + Entrar` (o usa el botón `Run`). Verás que todas esas líneas se ejecutan por orden. Por otra parte, no hace falta que seleccionemos todo el código. Este mecanismo te permite ejecutar parcialmente el código, simplemente seleccionando un bloque de líneas consecutivas, que se ejecutarán desde la primera a la última.

Ejercicio 21.
Prueba a ejecutar a la vez las líneas 5 *y* 7 *del fichero de código (y sólo esas líneas).* □

El *Editor de Código* también te permite modificar el fichero de instrucciones sobre el que estamos trabajando. Vamos a practicar esto.

Ejercicio 22.

1. *Cambia las líneas* 1 *y* 3 *por estas dos:*

```
(vector1 = 1:50 )

(vector2 = sample(1:50) )
```

   No te preocupes por el momento, pronto veremos en detalle como se utiliza la función `sample`. *Aquí se limita a ordenar al azar los números del* 1 *al* 50.

2. *Añade, en la línea* 11 *del* Editor de Código, *el comando:*

```
(varMuestral = var(vector3) )
```

   y ejecútalo desde el Editor de Código.

3. *Usa los menús* `File`→`Save`, *o el atajo de teclado* `Ctrl + S`, *para guardar el fichero modificado. Cierra RStudio, y abre el fichero con un* Editor de Texto. *Añade una línea al final del fichero (línea* 13, *por ejemplo) con el comando:*

```
boxplot(vector3)
```

   Guarda el fichero `Tut02-codigo01.R` *modificado, cierra el editor de texto, y vuelve a abrir el fichero en RStudio. Comprueba que aparecen todas las modificaciones que hemos ido haciendo, y ejecuta todos los comandos, al menos un par de veces.*

□

Las modificaciones que hemos hecho en este ejercicio no son importantes en sí mismas. El objetivo del ejercicio es demostrar que los ficheros de código de R son simplemente ficheros de texto, que se pueden manipular dentro del propio RStudio, pero también fuera de él, con un editor de texto cualquiera (aunque, como hemos dicho, RStudio ofrece muchas herramientas para hacer nuestro trabajo más cómodo).

Eso mismo sucede, en general, con los ficheros de código (a veces decimos *código fuente*) de cualquier lenguaje de programación (en inglés, esos ficheros se denominan *source code files*). Y, gracias a eso, resulta posible guardar nuestro trabajo muy fácilmente de una vez para otra, o intercambiar esos ficheros con otras personas. Si hemos hecho un análisis estadístico especialmente interesante, y creemos que puede resultar útil para otras personas que estudian problemas parecidos, basta con hacerles llegar nuestro fichero de código, para que puedan repetir ese análisis con sus datos.
Por ejemplo, el fichero adjunto:

Tut02-codigo02a.R,

contiene (casi todos) los comandos que hemos visto en la Sección 4, para hacer Estadística Descriptiva del vector `var3`. En la Tabla 2 puedes ver el contenido del fichero, que en el caso de ficheros de código llamaremos, a menudo, el listado del fichero.

```
setwd("")

df.var3 = read.table(file="./datos/Tut02-var3.csv")
var3 <- df.var3$V1

( minimo = min(var3) )
( maximo = max(var3) )
( rango = range(var3) )

( tablaFrecAbs = table(var3) )

( n = length( var3 ) )

( tablaFrecRel = tablaFrecAbs / n )
( tablaFrecAcu = cumsum(tablaFrecAbs)  )
( tablaFrecRelAcu = cumsum(tablaFrecRel) )


barplot(tablaFrecAbs, col = heat.colors(15))

boxplot(var3)

( media = mean(var3))

( varMuestral = var(var3))

( desvTipMuestral = sd(var3))

( varPobl= ( (n-1) / n ) * varMuestral )

( desvTipPobl = sqrt( varPobl ) )

( mediana = median(var3))

summary(var3)

( rangoIntCuart= IQR(var3))

( percentiles = quantile(var3, c(0.05, 0.15, 0.58, 0.75)) )
```

Tabla 2: Comandos de estadística descriptiva en el fichero `Tut02-codigo02a.R`

Fíjate en que, en la primera fila del fichero, en la línea en la que aparece la función `setwd`, hemos dejado vacío el espacio donde debería estar el nombre del directorio de trabajo (recuerda la discusión de la página 13, en la que aprendimos a fijar ese directorio). La razón es fácil de entender: si le enviamos este fichero a otra persona, mi directorio de trabajo no le servirá, y lo primero que tendrá que hacer es cambiar ese directorio de trabajo para que se corresponda con el directorio de su ordenador que vaya a jugar ese papel. Por otra parte, si se desea calcular otros percentiles, hay que modificar el vector de la última línea del código.

Evidentemente, cuando le enviamos el fichero a esa persona, vamos a tener que explicarle estos detalles, si es que queremos que el fichero le sirva de algo. A lo mejor te parece exagerado. Ten

en cuenta que este fichero, al ser de los primeros que vemos, es muy sencillo. Pero a medida que vayamos avanzando en el curso, los ficheros serán cada vez más complejos, y necesitarán cada vez más "instrucciones de manejo". Te podemos asegurar que llegará un momento, no muy lejano, en el que si escribes un fichero de código R, y dejas pasar 15 días sin usarlo, cuando lo vuelvas a abrir, pasados esos días, entonces tú mismo, por más que seas el autor del fichero, no recordarás para qué servía aquella variable, o porque hiciste las cosas así y no de otra manera, etc. Una de las verdades más contrastadas por la experiencia sobre la programación es esta:

El código sin documentación es inútil.

Ya dijimos, al hablar de Calc en el Tutorial-01, que había que acostumbrarse a identificar las operaciones con una descripción adecuada. En R eso se convierte en una necesidad mucho mayor. Así que, en el próximo apartado, vamos a empezar a aprender a documentar el código en R.

5.1. Comentarios en R.

Documentar significa, en el contexto de los ficheros de código R, añadir a esos ficheros información que no está pensada para dar instrucciones al ordenador, sino que ha sido pensada para ayudarnos a entender lo que se está haciendo en esa sesión de trabajo. Es decir, esa información no es para la máquina, es para nosotros mismos, o para otros usuarios (humanos) de ese fichero. A lo largo del curso, en los tutoriales, nosotros te vamos a facilitar una serie de ficheros (los llamaremos *"plantillas"*), que contienen código preparado para llevar a cabo algunos de los métodos que vamos a aprender en cada capítulo del curso. Cuando abras por primera vez uno de esos ficheros, y especialmente al tratarse de métodos con los que, aún, no estás familiarizado, necesitarás sin duda unas "instrucciones de manejo", para saber como utilizar el fichero. Esas instrucciones podrían ir en un fichero aparte, claro. Pero la experiencia ha demostrado que esa no es una buena manera de organizar el trabajo. Si el código y la documentación van por separado, es casi inevitable que, al final, tras algunos cambios, ya no se correspondan, y la documentación pase a ser contraproducente. Afortunadamente, hay una manera sencilla de combinar código y documentación. Fíjate en la segunda versión del fichero de comandos de Estadística Descriptiva,

Tut02-EstadDescriptivaNoAgrup-a.R

que aparece en la Tabla 3 (pág. 35). Como puedes ver, hemos añadido al fichero unas líneas nuevas, que empiezan todas con el símbolo `#`. Esas líneas son comentarios. Cuando R encuentra el símbolo `#` en cualquier línea de un fichero de código, deja de leer el resto de esa línea. Simplemente lo ignora, y pasa a leer la siguiente línea. Eso nos permite introducir nuestros comentarios, sin que interfieran con el trabajo de R. Si guardas el fichero en el directorio de trabajo y lo abres con RStudio verás que esas líneas aparecen en otro color, para distinguirlas de las líneas normales de código.

En el ejemplo de la Tabla 3 hemos añadido muchos comentarios, más de uno por cada línea de código. No siempre será necesario ni útil comentar todas y cada una de las líneas; ¡eso puede resultar, a veces, contraproducente! La combinación de unos nombres de variable bien elegidos, y la proporción justa de comentarios es la clave. Pero más vale pecar de exceso de comentarios que lamentar después su falta cuando, al cabo de un tiempo, no seamos capaces de entender el código; ni siquiera nuestro propio código. ¿Cuál es esa proporción justa de comentarios? No hay una respuesta única. Como siempre, la sabiduría te llegará (o no...) con el tiempo y la experiencia, cuando hayas visto el trabajo de los demás y desarrolles tu propio estilo. En cualquier caso, como poco deberías incluir unas líneas al principio del fichero explicando para qué sirve ese fichero, quién lo ha escrito, etc.

La segunda ventaja de utilizar comentarios, tal vez no resultará tan evidente al principio de tu trabajo con R. La posibilidad de tratar líneas del fichero como comentarios, y conseguir de esa forma que R las ignore por completo, nos permite *"activar"* o *"desactivar"*, de una forma muy sencilla, partes enteras de un fichero de código R antes de ejecutarlo. Por ejemplo, si cambiamos una línea de código

```
a=a+1
```

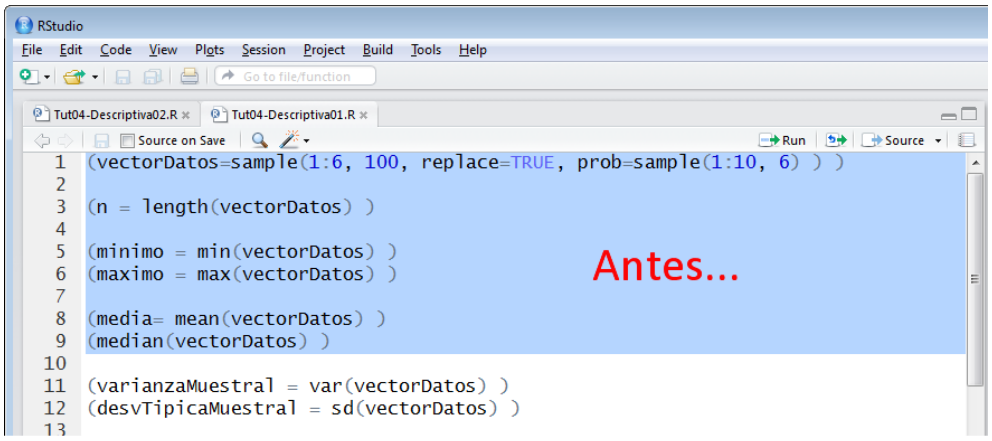
por esta otra

```
# a=a+1
```

```
#########################################################
# www.postdata-statistics.com
# POSTDATA. Introducción a la Estadística
# Tutorial 02.
# Plantilla de comandos R para Estadística Descriptiva
# Una variable cuantitativa, datos no agrupados.
#########################################################


# ATENCION: para empezar a trabajar es necesario establecer
# el directorio de trabajo en la siguiente línea de código.
# y el nombre del fichero (más opciones) en la línea 18.
# De lo contrario este fichero no funcionará.
setwd("")

# Leemos el fichero de datos, y lo guardamos en la variable vectorDatos.
# El fichero debe estar en la subcarpeta datos del directorio de trabajo.
df.Datos = scan(file="./datos/")
vectorDatos <- df.Datos$V1

# Calculamos máximo, minimo y rango.
( minimo = min(vectorDatos) )
( maximo = max(vectorDatos) )
( rango = range(vectorDatos) )

# Determinamos la longitud del vector de datos.
( n = length( vectorDatos ) )

# Hallamos las tablas de frecuencias:
# (1) absoluta,  (2) relativa,
# (3) acumulada, (4) acumulada relativa.
( tablaFrecAbs = table(vectorDatos) )
( tablaFrecRel = tablaFrecAbs / n )
( tablaFrecAcu = cumsum(tablaFrecAbs)  )
( tablaFrecRelAcu = cumsum(tablaFrecRel) )

# Dibujamos un gráfico de barras de las frecuencias.
barplot(tablaFrecAbs, col = heat.colors(15))

# Y el diagrama de caja.
boxplot(vectorDatos)

# Calculo de la media aritmética,
( media = mean(vectorDatos))

# la cuasivarianza muestral,
( varMuestral = var(vectorDatos))

# y la cuasidesviación típica.
( desvTipMuestral = sd(vectorDatos))

# La varianza y desviación típica poblacionales
# se obtienen así:
( varPobl= ( (n-1) / n ) * varMuestral )
( desvTipPobl = sqrt( varPobl ) )

# Calculamos la mediana.
( mediana = median(vectorDatos))

# La función summmary muestra la media y varias
# medidas de posición (cuartiles).
summary(vectorDatos)

# El rango intercuartílico.
( rangoIntCuart= IQR(vectorDatos))

# Y algunos percentiles: 5%, 15%, 58% y 75%.
# Si deseas otros percentiles, modifica los
# valores del vector.

( percentiles = quantile(vectorDatos, c(0.05, 0.15, 0.58, 0.75)) )
```

Tabla 3: Añadiendo comentarios, fichero Tut02-EstadDescriptivaNoAgrup-a.R. Primera "plantilla" de comandos R del curso.

no estaremos "explicando" nada, sino que estamos cambiando el comportamiento de R, y evitando que esa línea en particular se ejecute. Y hacer ese cambio sólo nos cuesta el trabajo de introducir un símbolo `#`. Esto permite tener un único fichero de instrucciones R, que se puede comportar de distinta manera, jugando con los símbolos de comentario. Veremos ejemplos de esto más adelante, y comprobaremos lo práctico que puede llegar a ser.
Cuando hacemos esto, es decir cuando añadimos `#` al comienzo de una línea de código R, decimos que hemos *comentado* esa línea (y la *descomentamos* cuando quitamos el `#`). Esta operación es bastante frecuente, así que RStudio te ofrece una herramienta, llamada `Comment/Uncomment lines` en el menú `Code`. Si seleccionas una o varias líneas de código, y usas esta opción, verás que RStudio añade `#` al comienzo de todas las líneas comentadas.



No podemos exagerar la importancia de que te acostumbres, desde el primer momento, a incorporar la documentación como una parte inseparable de tu trabajo con R. Te lo recordaremos a menudo, especialmente en los primeros tutoriales. Aparte de esto, la idea de combinar código y documentación se puede llevar, en R, mucho más lejos de lo que aquí hemos visto. Aunque no vamos a poder ocuparnos de ellas en este curso, mediante herramientas como *R Markdown*, *Sweave* o *Knitr*, entre otras, es posible obtener documentos en los que el código R se combina con una documentación mucho más rica, que en algunos casos puede incluir figuras, fórmulas matemáticas complejas, etc.

6. Más operaciones con vectores en R.

Los vectores son un ingrediente esencial de nuestro trabajo con R. En esta sección vamos a (empezar a) aprender cómo fabricar vectores a la medida de nuestros deseos, y a manipular esos vectores con precisión.

6.1. Números aleatorios: la función sample de R.

En el Tutorial-01 vimos como generar números (pseudo)aleatorios con Calc. Y en el Capítulo 3 de la teoría del curso se usan esos números para hacer varios experimentos relacionados con las probabilidades, en situaciones bastante elementales. Para prepararnos, vamos a aprender a hacer lo mismo con R. Como veremos, en este y en futuros tutoriales, se hará evidente que R es un especialista en la materia y, usándolo, vamos a poder ir mucho más allá de lo que resulta viable hacer con Calc.

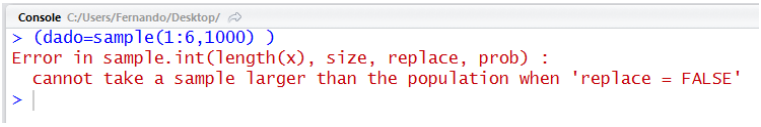
El primer paso, para empezar a generar números aleatorios con R es utilizar la función `sample`. En su uso más elemental, esta función toma un vector de datos (el que nosotros queramos) y permite elegir al azar k elementos del vector, con o sin reemplazamiento. Y no se agotan ahí sus capacidades, así que vamos a ir viendo diversos ejemplos.
Empezamos por lo más simple, lanzar un dado. Para eso ejecutamos este código:

```
(dado=sample(1:6, size=1) )
```

Algunas cosas ya nos resultan conocidas. Vamos a guardar el resultado en una variable que hemos llamado `dado`, y hemos rodeado el comando con paréntesis para que R, además de hacer las operaciones que pedimos, nos muestre el resultado. Prueba a ejecutar este comando varias veces seguidas, para ver como, en cada lanzamiento, obtienes un resultado distinto. Naturalmente, esto no sirve si lo que quieres es lanzar el dado muchas veces, pongamos mil veces. Puedes pensar que bastaría con escribir

```
(dado = sample(1:6, size=1000) )
```

Prueba a ejecutar este comando, y verás que R te obsequia con un bonito mensaje de error:



El mensaje nos explica cual ha sido el error: hemos tratado de extraer 1000 elementos de un vector que sólo tiene seis, y eso no es posible si `replace=FALSE`, es decir, si el muestreo es sin reemplazamiento. El remedio está claro, hacemos (recuerda que puedes usar el tabulador para ayudarte):

```
(dado = sample(1:6, size=1000, replace = TRUE) )
```

y, ahora sí, verás desfilar 1000 lanzamientos del dado por tu pantalla. Fíjate en que la variable `dado`, que es el resultado de `sample`, es un vector.

Ejercicio 23.

1. *Obtén la tabla de frecuencias, y calcula la media aritmética y la cuasidesviación típica (muestral) de los valores que hay en el vector* `dado`.
2. *¿Cómo puedes comprobar que la variable* `dado` *es un vector?*

Solución en la página 52. □

¿Y si en lugar de lanzar un dado mil veces queremos sacar cinco cartas (sin reemplazamiento) de una baraja de 48 cartas? Podemos representar las cartas de la baraja con los números del 1 al 48,y entonces basta con hacer:

```
(carta = sample(1:48, size=5, replace=FALSE) )
```

Hasta ahora, hemos visto ejemplos en los que extraíamos valores aleatorios de rangos (o intervalos), como `1:6`, y `1:48`. Naturalmente, los rangos no tienen porque empezar en 1, podemos aplicar `sample` a un rango como `161:234`. Otras veces, en cambio, nos sucederá que tenemos un vector, como este vector `edades`:

```
edades = c(22, 21, 18, 19, 17, 21, 18, 20, 17, 18, 17, 22, 20, 19, 18, 19, 18,
                                    22, 20, 19)
```

y lo que queremos es extraer algunos de estos valores al azar; pongamos por ejemplo, que queremos extraer 7 elementos. Para hacer eso, si queremos muestreo sin reemplazamiento basta con usar:

```
(edadesAlAzar = sample(edades, size=7, replace=FALSE) )
```

y si lo queremos con reemplazamiento pondremos `replace=TRUE`. Fíjate en que, en este caso, el vector original ya contiene elementos repetidos. Así que, independientemente de que el muestreo sea con reemplazamiento o sin él, al usar sample podemos obtener valores repetidos. Lo que sample sortea son las *posiciones* dentro del vector `edades`, y no los *valores* que ocupan esas posiciones. Para entender esto un poco mejor, prueba a ejecutar este código, en el que usamos `sample` sobre un vector que tiene el número 1 repetido 9 veces y un único 2:

```
muchosUnos = c(1,1,1,1,1,1,1,1,1,2)
(muestra = sample(muchosUnos, size=100, replace=TRUE) )
```

Antes de pasar a otras cosas, un truco adicional con la función `sample`. Si aplicas `sample` a un vector, sin ningún argumento más, obtendrás todos los elementos del vector, pero en un orden aleatorio (en lenguaje más técnico, una permutación aleatoria de los elementos del vector). En seguida vamos a ver la función `sort`, que es, en algún sentido, lo contrario de `sample`. La función `sort` ordena, y `sample` desordena. Prueba, por ejemplo a ejecutar varias veces:

```
sample(1:20)
```

para ver el resultado.

Funciones `unique` y `sort`

A la vista de los últimos ejemplos, y ahora que estamos aprendiendo a fabricar un vector con muchos elementos repetidos, es conveniente subrayar que a veces necesitaremos saber cuáles son los elementos *distintos* que contiene ese vector. El recorrido (rango) del vector nos dice el máximo y el mínimo, pero desde luego no nos dice cuáles de los valores intermedios aparecen, de hecho, en el vector. Una manera de obtener esto es mediante la tabla de frecuencias, pero en ese caso el resultado es una tabla, no un vector, y a veces es más difícil de manipular. Afortunadamente, en R tenemos soluciones para todo, y es muy sencillo obtener la respuesta con la función `unique`. Por ejemplo, para el vector `edades` que hemos usado antes, es:

```
> (edadesDistintas=unique(edades))
[1] 22 21 18 19 17 20
```

Como ves, R devuelve un vector con los valores distintos que hay en `edades`, pero en el orden en que los va encontrando, y no hace ningún esfuerzo por ordenarlos (puede que nos interese conservar el orden, así que eso esta bien).
"¡Pero es que yo los quiero ordenados!" ¿Hemos dicho ya que en R hay soluciones para todo? Usamos la función `sort`:

```
> (edadesDistintasEnOrden=sort(unique(edades)))
[1] 17 18 19 20 21 22
```

"Es que las quiero ordenadas en orden decreciente..."

```
> (edadesDistintasEnOrden=sort(unique(edades), decreasing = TRUE ))
[1] 22 21 20 19 18 17
```

Una aclaración importante sobre `sort`. La ordenación no afecta en absoluto al vector original. Si quieres ordenar un vector, por ejemplo llamado `datos` y guardar el vector ordenado con el mismo nombre, entonces puedes usar el comando

```
datos = sort(datos)
```

pero es importante entender que este es un viaje sin retorno: el vector `datos` original, sin ordenar, se habrá perdido, y sólo tendremos la versión ordenada. ¡Cuidado con la posible pérdida de información relevante!
La función `sort` sirve, desde luego, para ordenar valores numéricos. Pero también hace ordenación alfabética. Vamos a aprovechar para presentarte un par de vectores especiales, los vectores `letters` y `LETTERS`, que vienen predefinidos en R:

```
> letters
 [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q" "r"
[19] "s" "t" "u" "v" "w" "x" "y" "z"
> LETTERS
 [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L" "M" "N" "O" "P" "Q" "R"
[19] "S" "T" "U" "V" "W" "X" "Y" "Z"
```

Como ves, esos vectores contienen las letras del alfabeto (para el idioma inglés). Así que podemos fabricar un vector de letras al azar (¿un generador de contraseñas?) con este código:

```
> ( Clave = sample(LETTERS, size=15, replace=TRUE) )
 [1] "U" "Z" "T" "H" "I" "J" "E" "X" "J" "Y" "P" "G" "V" "P" "P"
```

Como ves, es un generador de contraseñas muy rudimentario. No te preocupes, en breve podremos hacer uno tan sofisticado como sea preciso. ¿Cuántas letras *distintas* aparecen en esa clave? Con las funciones `unique` y `sort` podemos conseguir la respuesta ordenada alfabéticamente:

```
> sort(unique(Clave))
 [1] "E" "G" "H" "I" "J" "P" "T" "U" "V" "X" "Y" "Z"
```

Ejercicio 24.
Modifica el generador de claves para que utilice mayúsculas, minúsculas, y cifras. Indicación: recuerda que `c` *es por* concatenate. *Solución en la página 52.* □

Números pseudoaleatorios, pero "reproducibles": la función `set.seed`.

Al principio de esta Sección 6.1 hemos lanzado 1000 veces un dado, con el comando

```
(dado = sample(1:6, size=1000, replace = TRUE) )
```

El inconveniente de trabajar con números aleatorios es que los resultados del lector serán diferentes de los nuestros y, de hecho, serán diferentes cada vez que ejecutes la función `sample`. Los números aleatorios se utilizan mucho, por ejemplo, para hacer simulaciones. Y si queremos hacer una de esas simulaciones, y compartirla con otras personas, de manera que puedan *verificar* nuestros resultados, entonces necesitamos:

- Que los números sean aleatorios, en el sentido de que nosotros no los hemos elegido, sino que son el resultado de un *sorteo*.
- Pero que los resultados del sorteo queden registrados de alguna manera, para que otros puedan reproducirlos.

Afortunadamente (en este caso), como ya dijimos, los números que produce un ordenador no son aleatorios, sino pseudoaleatorios. Y para lo que aquí nos ocupa, eso es una ventaja. Hay una función de R, llamada set.seed, que permite decirle a R que queremos hacer exactamente esto: generar números aleatorios reproducibles. Concretamente, para ver funciona como esto, prueba en primer lugar a ejecutar varias veces el comando

```
sample(1:100, size=15, replace=TRUE)
```

Cada vez obtendrás una lista de números diferente. Prueba ahora a ejecutar estos dos comandos:

```
set.seed(2014)
sample(1:100, size=15, replace=TRUE)
```

La respuesta será

```
> set.seed(2014)
> sample(1:100, size=15, replace=TRUE)
 [1] 29 17 63 31 55  9 92 61 10 16 63  6 61 59 66
```

Y será la misma en tu ordenador, en el mío, o en el de cualquier otra persona que ejecute R. Como ves, la función `set.seed` utiliza un argumento, al que llamamos la semilla (en inglés, *seed*), que en este caso yo he fijado, arbitrariamente, en 2014. La idea es que si utilizas `set.seed` con la misma semilla que yo, obtendrás los mismos números pseudoaleatorios que yo he obtenido.
Una vez visto lo fundamental, no queremos entretenernos mucho más en esto. Pero no podemos dejar de mencionar que el asunto de cómo se elige la semilla es delicado. Podría parecer que lo mejor, en una simulación, es elegir la propia semilla "al azar". El problema es que, de esa manera, en caso de que alguien sospeche que se han manipulado los datos, puede pensar que hemos ido probando varias de estas semillas "al azar", hasta obtener unos resultados especialmente buenos de la simulación. Muchos autores recomiendan, como alternativa, fijar una política con respecto a la elección de la semilla, y atenerse a ella en todas las simulaciones. Por ejemplo, puedes usar siempre como semilla el año en que realizas la simulación, como hemos hecho aquí.

6.2. Vectores a medida con `seq` y `rep`.

Ya sabemos que, en R, podemos crear un vector con los números del 1 al 100, basta con usar `1:100`. Pero habíamos dejado pendiente el problema de conseguir una forma más flexible de fabricar los vectores. ¿Y si queremos, por ejemplo, empezar en 1 y avanzar de 3 en 3, sin pasar de 100? Es decir, queremos fabricar los números:

$$1,\ 4,\ 7,\ \ldots,\ 94,\ 97,\ 100.$$

Esto se puede conseguir en R de muy fácilmente, usando la función `seq` (de *sequence*, secuencia o sucesión, en el sentido matemático del término). Hacemos:

```
( deTresEnTres = seq(from =1, to=100, by=3) )
```

y obtenemos como respuesta:

```
   > ( deTresEnTres = seq(from =1, to=100, by=3) )
 [1]   1   4   7  10  13  16  19  22  25  28  31  34  37  40
[15]  43  46  49  52  55  58  61  64  67  70  73  76  79  82
[29]  85  88  91  94  97 100
```

(Ya sabes que el formato de salida y los números entre corchetes dependen de la anchura de tu pantalla). La función `seq` permite fabricar vectores cuyos elementos se diferencian en una cantidad fija; que puede ser cualquier número, entero, fraccionario, expresado con decimales, etc.

Recuerda que los argumentos de las funciones de R se pueden escribir con nombre, como hemos hecho aquí (son `from, to, by`). Pero, si sabes lo que haces, también puedes usar:

```
( deTresEnTres = seq(1, 100, 3) )
```

En cualquier caso, al principio conviene que seas prudente con esto (y recuerda que el tabulador es tu amigo en RStudio).

La función `seq` es, además, muy útil para dividir un intervalo (a, b) en n subintervalos iguales (esto te vendrá bien, por ejemplo, cuando quieres dividir los valores de una variable cuantitativa continua en clases). Podríamos hacer las cuentas "a mano", calculando la longitud de los subintervalos, etc. Pero `seq` facilita mucho el trabajo, remplazando la opción `by` con la opción `length.out`, que debes interpretar como *cuántos valores quieres que contenga el vector de salida*. Por ejemplo, si quiero dividir el intervalo $(15, 27)$ en 19 subintervalos iguales, teniendo en cuenta que se incluyen los extremos, está claro que necesito $19 + 1 = 20$ puntos en el vector de salida. En este tipo de situaciones es típico usar el nombre `nodos` para los extremos de los subintervalos que vamos a fabricar (no los confundas con las marcas de clase, que son los puntos medios de esos subintervalos). Así que hacemos

```
(nodos = seq(from=15 , to=27 , length.out=20 ) )
```

La respuesta es justo lo que queríamos:

```
> (nodos = seq(from=15 , to=27 , length.out=20 ) )
 [1] 15.00000 15.63158 16.26316 16.89474 17.52632 18.15789 18.78947 19.42105
 [9] 20.05263 20.68421 21.31579 21.94737 22.57895 23.21053 23.84211 24.47368
[17] 25.10526 25.73684 26.36842 27.00000
```

Hay otra función, la función `rep` (de *repeat*, repetir), estrechamente emparentada con `seq`, cuyo principal uso es fabricar vectores, siguiendo un patrón que nosotros definimos. El nombre `rep` viene de repeat, repetir. Y eso ayuda a adivinar lo que sucede si escribimos este comando:

```
valores = c(1,2,3,4,5)
rep(valores, times=3)
```

El argumento `times` (en inglés, aquí con significado de *"veces"*). La salida correspondiente es, como cabría esperar, el vector datos repetido tres veces:

```
> valores=c(1,2,3,4,5)
> rep(valores, times=3)
[1] 1 2 3 4 5 1 2 3 4 5 1 2 3 4 5
```

Por cierto, ese argumento opcional `times` es uno de los raros casos que no se obtiene con el tabulador en RStudio.

Quizá sea un poco menos fácil adivinar lo que sucede con este otro comando:

```
valores = c(1, 2, 3, 4, 5)
rep(valores, times=c(2, 9, 1, 4, 3) )
```

Si lo ejecutamos, la salida es:

```
> valores = c(1, 2, 3, 4, 5)
> rep(valores, c(2, 9, 1, 4, 3) )
[1] 1 1 2 2 2 2 2 2 2 2 2 3 4 4 4 4 5 5 5
```

Y ahora está más claro lo que sucede: el segundo argumento le dice a R cuántas veces debe repetir cada uno de los correspondientes elementos del vector datos (el primer argumento). Aquí el vector que usamos en `times` juega el papel de *vector de frecuencias* para los elementos del vector `valores`. Por eso, este funcionamiento de rep es especialmente interesante cuando tenemos que empezar a trabajar a partir de datos representados mediante tablas de frecuencias. Para ver un ejemplo sencillo, vamos a suponer que tenemos dos vectores:

```
valores = c(2,3,5,8,13)
frecuencias = c(5,7,12,2,14)
```

de manera que el primero contiene los valores (distintos) que aparecen en nuestros datos, y el segundo contiene las frecuencias de esos valores (por supuesto, los dos vectores tendrán la misma longitud). Entonces, para recuperar el conjunto inicial de datos (sin agrupar) partiendo de esta "tabla de frecuencias", podemos usar así la función `rep`:

```
( datosSinAgrupar = rep(valores, times=frecuencias) )
```

Y se obtiene este resultado:

```
> ( datosSinAgrupar = rep(valores, frecuencias) )
 [1]  2  2  2  2  2  3  3  3  3  3  3  3  5  5  5  5  5  5  5  5  5  5  5
[25]  8  8 13 13 13 13 13 13 13 13 13 13 13 13 13 13 13
```

Ahora puedes utilizar las funciones habituales para hacer la Estadística Descriptiva del vector `datosSinAgrupar`. Esta forma de "desempaquetar" tablas de frecuencia nos será útil en ocasiones. En este último ejemplo hemos usado una "'tabla de frecuencias" que en realidad no es una tabla, sino dos vectores por separado, uno con los valores y otro con las frecuencias. Para ver con más claridad, el fichero csv tut04-EjemploTablaFrecuencias.csv contiene esta misma tabla de frecuencias.

6.3. Seleccionando elementos de vectores.

La notación de corchetes. Funciones `head` y `tail`.

Volvemos entonces a nuestros 1000 lanzamientos del dado, pero ahora vamos a hacer:

```
set.seed(2014)
dado = sample(1:6, size=1000, replace = TRUE)
```

De esa forma, el lector y nosotros trabajaremos sobre el mismo vector `dado`, y podrás comparar tus resultados con los nuestros. En el Ejercicio 23 (pág. 37) hemos invitado al lector a que hiciera la tabla de frecuencias del vector `dado`, resultante de ese experimento. El resultado de aquel ejercicio (para el vector `dado` que hemos fabricado después de usar `set.seed`) es:

```
> table(dado)
dado
  1   2   3   4   5   6
145 154 167 192 167 175
```

Eso nos permite, por ejemplo, ver que el número 5 ha aparecido 167 veces. Pero no nos permite saber cuál fue el resultado de la tirada número 23. Para averiguar esto, en R escribimos

```
> dado[23]
[1] 4
```

y vemos que en esa tirada se obtuvo un 4. De esa forma, usando los corchetes, podemos acceder directamente a cualquier posición del vector. Por ejemplo, el primer y el último elemento del vector son:

```
dado[1]
dado[length(dado)]
```

En este caso sabemos que el vector tiene 1000 elementos, y para obtener el último elemento podríamos haber escrito

```
dado[1000]
```

pero la otra versión (usando length) tiene la ventaja de que seguirá funcionando aunque el vector cambie. Si quieres ver unos cuantos valores del principio (respectivamente del final) del vector, puedes usar `head(dado)` (respectivamente, `tail(dado)`). Estas dos funciones, por defecto, te mostrarán los últimos seis elementos del vector, pero puedes usar el argumento opcional `n` para controlar cuántos se muestran.

Ejercicio 25.
Usa la función `tail` *para obtener el último elemento del vector* `dado`. *Solución en la página 52.* □

Aunque las funciones `head` y `tail` son útiles, la herramienta básica para acceder a los elementos de un vector son los corchetes `[]` (en inglés, *(square) brackets*). Este recurso de los corchetes es mucho más potente de lo que parece a primera vista. Podemos seleccionar varios elementos a la vez indicando sus posiciones. Para seleccionar los elementos de dado en las posiciones de la 12 a la 27 hacemos:

```
> dado[12:27]
 [1] 1 4 4 4 3 5 1 6 2 6 5 4 5 4 6 3
```

y obtenemos un vector con los números que ocupan exactamente esas posiciones. Y si las posiciones que queremos no son consecutivas, basta con indicarlas mediante un vector (no olvides la `c` al crear el vector). Por ejemplo:

```
> dado[c(3,8,21,43,56)]
[1] 4 4 6 6 1
```

nos proporciona los elementos en esas posiciones del vector. Cuando se combina esto con otras ideas, que estamos viendo en este tutorial, se pueden conseguir resultados muy útiles. Por ejemplo, ¿qué crees que sucede al hacer esto? (Haz el experimento).

```
datos=100:200
datos
datos[seq(1,length(datos),3)]
```

Selección de elementos mediante condiciones. Valores booleanos. Función `which`.

Pero aún podemos ir más lejos. Podemos utilizarlo para seleccionar los elementos del vector que cumplen cierta condición. Por ejemplo, queremos seleccionar los resultados, en el vector `dado`, que son mayores que 3. En primer lugar, igual que aprendimos (en el anterior tutorial) a sumar una cantidad a un vector, o a elevar un vector al cuadrado, etc., podemos también aplicar una condición lógica, como una desigualdad, a un vector (¡no ejecutes el comando todavía!):

```
dado > 3
```

Tienes que aprender a leer esta expresión como si fuera una pregunta: "¿Es el resultado de lanzar el dado mayor que 3?". La respuesta sólo puede ser "sí" o "no", o dicho de otra manera "cierto" o "falso" [1]. En R (y en muchos lenguajes de programación), las respuestas a ese tipo de preguntas se representan con un tipo especial de variables, las variables booleanas (por el matemático G. Boole[2]). Una variable de tipo booleano, por tanto, sólo puede tomar dos valores, que son `TRUE` (cierto) o `FALSE` (falso). Las variables de tipo booleano son esenciales en programación. Un programa tiene que tomar muchas decisiones del estilo "si la condición C es cierta, haz la acción A, y si no, haz la acción B". Esas decisiones dependen del resultado de la "condición C", que siempre es una variable booleana. Veremos ejemplos, usando R, en futuros tutoriales.
¿Cuál es el resultado de `dado < 3`? Pues un vector de 1000 valores booleanos. Para ver los 10 primeros vamos a hacer:

```
> head( dado < 3, n=10)
 [1]  TRUE  TRUE FALSE  TRUE FALSE  TRUE FALSE FALSE  TRUE  TRUE
```

Y para que puedas comprobar de donde proceden esos valores, aquí tienes los 10 primeros elementos de `dado`.

```
> head( dado , n=10)
 [1] 2 2 4 2 4 1 6 4 1 1
```

Cada elemento del vector booleano que se obtiene con `dado < 3` nos dice si la condición "es mayor que 3" es cierta o falsa en la correspondiente posición de `dado`.
Las cosas empiezan a ponerse divertidas cuando combinamos esta idea con la selección por corchetes, por ejemplo, así:

```
mayoresQue3 = dado[ dado > 3 ]
```

Entonces, en el vector `mayoresQue3` obtenemos un vector que contiene precisamente aquellos elementos de `dado` que eran mayores que 3 (por lo tanto, que valen 4, 5 o 6):

```
> mayoresQue3
  [1] 4 4 6 4 4 4 4 4 5 6 6 5 4 5 4 6 6 5 6 5 4 5 5 5 4 6 6 4 5 4 5 6 5 6 4
 [36] 4 6 5 4 6 5 5 4 4 5 5 5 5 6 6 4 6 4 4 5 4 5 6 4 6 5 5 5 6 4 4 4 4 4 6
 [71] 6 4 6 4 4 6 6 6 5 4 4 6 4 5 5 4 6 4 4 4 6 4 5 6 4 6 6 5 5 5 5 5 4 5 6
[106] 5 4 6 5 5 5 6 4 6 4 6 6 4 4 4 5 4 5 6 4 4 5 6 6 5 4 5 6 5 4 5 4 6 4 5
[141] 5 6 6 4 4 6 4 4 4 6 5 5 6 4 6 4 4 6 5 6 4 4 6 6 5 6 5 5 6 6 4 4 6 4 5
[176] 6 4 5 6 6 4 6 4 5 4 5 4 6 4 5 5 4 5 5 4 6 6 6 5 6 4 4 4 4 4 4 5 5 6 6
[211] 6 4 6 6 6 4 5 6 4 5 5 4 6 6 4 5 5 4 4 4 4 4 6 6 4 5 5 6 5 6 4 6 6 6 4
[246] 6 6 5 5 5 6 6 5 5 4 4 5 4 4 6 6 6 5 6 4 6 6 6 4 6 6 5 4 5 5 6 6 6 5 4
[281] 5 4 4 6 5 5 6 4 4 5 5 6 4 4 5 6 6 6 6 4 6 5 4 4 6 6 5 6 6 4 4 6 6 6 4
[316] 6 5 4 6 5 5 4 6 5 5 6 6 4 6 4 6 5 6 5 4 5 4 4 6 6 6 6 4 4 4 5 4 6 6 6
[351] 6 5 4 6 6 6 6 5 6 6 5 6 4 6 4 5 4 5 6 5 6 5 6 5 4 5 5 5 5 6 4 4 5 6 5
[386] 5 5 4 4 5 4 4 6 4 6 4 5 5 4 5 5 5 4 6 5 4 4 6 6 6 4 5 5 5 6 5 4 6 4 4
[421] 5 6 5 6 6 5 5 5 4 6 6 4 4 4 6 4 5 5 6 4 4 4 5 4 5 6 4 5 5 5 4 5 5 5 6
[456] 5 6 6 5 4 6 6 4 6 5 5 5 6 6 5 4 4 4 5 4 5 5 6 5 4 6 4 4 6 5 5 5 4 4 6
[491] 4 4 5 5 4 4 4 6 4 4 4 6 4 4 5 4 4 4 6 4 5 6 5 6 5 4 4 5 5 5 5 5 4 4 4
[526] 6 4 4 5 4 6 4 5 4
```

Pero claro, en este vector hemos perdido la información sobre las posiciones originales que ocupaban esos valores en `dado`. Si quieres obtener esa información, usa la función `which` junto con la condición `dado > 3`:

```
which( dado > 3 )
```

Y obtendrás un vector de posiciones (sólo se muestra el principio, usando `head`):

```
> head( which( dado > 3 ), n=20)
 [1]  3  5  7  8 11 13 14 15 17 19 21 22 23 24 25 26 30 31 35 36
```

Para que quede claro: si miras el vector `dado`, en una de estas posiciones, encontrarás un valor mayor que 3. Y viceversa si la posición no aparece en esta lista, entonces el correspondiente valor de `dado`, será 3 o menos.

[1] en realidad hay un tercer caso, del que ya hablaremos, cuando el elemento del vector está ausente, como los valor `NaN` que ya hemos visto.
[2] Más información en http://es.wikipedia.org/wiki/George_Boole

Ejercicio 26.

1. *Comprueba que en la posición número* 16 *de* `dado` *(que no aparece arriba), hay un elemento menor o igual que* 3.
2. *Construye un vector que contenga las posiciones de* `dado` *que contienen números mayores o iguales que* 2.
3. *Un poco más difícil. Construye un vector que contenga las posiciones de* `dado` *que contienen números pares. Pista: ¿qué hacía el operador* `%%` *(ver el Ejercicio 1, pág. 4)?*

Solución en la página 52. □

Como puedes ver, nuestra capacidad de fabricar vectores a medida, muestrearlos (samplearlos) y seleccionar sus elementos, es muy grande. ¡Y ni siquiera hemos hablado aún de la función `paste`, que es probablemente una de las herramientas más versátiles para generar vectores en R! Esta función es extremadamente útil en Diseño Experimental, y hablaremos de ella en alguno de los próximos tutoriales.

Los valores booleanos como unos y ceros. Operadores booleanos.

Ya hemos dicho que los valores booleanos `TRUE` y `FALSE` son la base sobre la que se construye toda la toma de decisiones, en R y, en general, en cualquier programa de ordenador. Las decisiones se toman comprobando si se cumple una cierta condición, que, a veces, puede ser tan sencilla como las que hemos visto:

```
dado > 3
```

En cuanto esa condición sea un poco más complicada, necesitaremos más herramientas. Por ejemplo, si queremos seleccionar los elementos del vector `dado` que son, *a la vez*, más grandes que 3 y menores que 6, entonces necesitamos aprender una forma de expresar la conjunción "Y" en el lenguaje de R. En ese lenguaje, esto se expresa así:

```
(dado > 3) & (dado < 6)
```

El operador `&` es, en R, el Y booleano (en inglés se escribe a menudo *AND*, en mayúsculas). En otros lenguajes de programación se escribe de diferentes maneras. Para ver como funciona el operador `&`, vamos a ejecutar varios comandos:

```
> TRUE & TRUE
[1] TRUE
> TRUE & FALSE
[1] FALSE
> FALSE & TRUE
[1] FALSE
> FALSE & FALSE
[1] FALSE
```

Estos cuatro comandos cubren todas las situaciones posibles al usar `&`. Por ejemplo, al evaluar una condición como:

```
(3 < 5) & (6 < 4)
```

el primer paréntesis da como resultado `TRUE` y el segundo `FALSE`, así que (estamos en el segundo caso y) el operador `&` produce `FALSE`:

```
> (3 < 5) & (6 < 4)
[1] FALSE
```

Como ves, para que el resultado de `&` sea `TRUE` es necesario que ambos operandos sean `TRUE`. Otro operador booleano de R, estrechamente emparentado con `&`, es el operador `|` (una barra vertical), llamado O booleano (en inglés se escribe a menudo *OR*. Todo lo que se necesita saber de este operador se resume en estas cuatro situaciones posibles:

```
> TRUE | TRUE
[1] TRUE
> TRUE | FALSE
[1] TRUE
> FALSE | TRUE
[1] TRUE
> FALSE | FALSE
[1] FALSE
```

con la que no deberías tener problemas en hacer el siguiente:

Ejercicio 27.
¿Cuál es el resultado de la siguiente operación?

```
(7 < 5) | (2 < 4)
```

Solución en la página 52. □

Como ves, el comportamiento de `|` es simétrico al de `&`. Para que el resultado de `|` sea `FALSE` es necesario que ambos operandos sean `FALSE`.
R utiliza un sistema de equivalencia entre los valores booleanos `TRUE` y `FALSE` por un lado, y los números `1` y `0` por otro, que resulta a menudo extremadamente útil. Si le pedimos a R que *multiplique* `TRUE` por `FALSE`, lo que hará será cambiar `TRUE` por 1, `FALSE` por 0, multiplicar, obteniendo un 0:

```
> TRUE * FALSE
[1] 0
```

Puedes hacer lo mismo para hacer cualquier operación con los valores lógicos. Por ejemplo,

```
> TRUE/FALSE
[1] Inf
> TRUE + TRUE
[1] 2
> TRUE + FALSE
[1] 1
```

Los dos últimos resultados permiten usar los valores booleanos para *contar* los elementos de un vector que verifican una condición. En efecto, hemos visto que con la condición

```
dado > 3
```

se obtiene como respuesta un vector booleano de valores `TRUE` y `FALSE` que nos dicen, para cada elemento del vector `dado`, si esa condición es cierta o no. Para saber cuántos elementos la cumplen, es decir, cuántos valores `TRUE` hay en el vector booleano `dado > 3`, basta con tener en cuenta que los valores `TRUE` cuentan como 1 y los `FALSE` como 0, y *sumar* ese vector booleano:

```
> sum(dado >3)
[1] 534
```

Si tienes problemas para entender esto, piensa que el resultado de `dado > 3` es un vector de unos y ceros, y que lo que queremos es saber cuántos unos contiene.
Hemos visto que R convierte `TRUE` en 1 y `FALSE` en 0. Es bueno saber que R también hace conversiones en sentido contrario. Si usas los operadores lógicos `&` y `|` para operar *con números*, entonces el número 0 se convierte en `FALSE`, mientras que *cualquier otro número* se convierte en `TRUE`. Por ejemplo:

```
> 3 & 5
[1] TRUE
> 3 & 0
[1] FALSE
> 0 | 2
[1] TRUE
> 0 | 0
[1] FALSE
> -2.3 & 0
[1] FALSE
```

El último ejemplo pretende ilustrar que las mismas reglas funcionan con números fraccionarios o negativos. Recuerda: cualquier cosa que no sea un 0 se convierte en `TRUE`.

El operador "igual a"

Hemos visto cómo seleccionar los elementos del vector `dado` que son mayores que 3. Pero ¿y si necesitamos seleccionar aquellos que son exactamente iguales a 3? Tu primera reacción tal vez sea escribir

```
dado[dado=3]
```

pero eso no funcionará como esperas, porque en R, el símbolo `=` indica, como ya sabemos, una *asignación* de una variable a un valor. El significado de `=` es más bien `usa esta etiqueta para este valor`. Y lo que ahora necesitamos es una forma de decidir si dos valores son iguales. El operador adecuado, en R, se representa con dos símbolos igual concatenados:

```
> 3 == 5
[1] FALSE
> 3 == 3
[1] TRUE
```

y, al igual que sucede con `>`, o con `<`, es un operador booleano, que produce como resultado `TRUE` o `FALSE` (o `NA` si alguno de los argumentos es igual a `NA`).
Volviendo a la pregunta inicial, la forma de localizar los elementos de `dado` que son iguales a 3 es

```
dado[dado == 3]
```

Ejercicio 28.
Ejecuta estas instrucciones:

```
set.seed(2014)
dado2 = sample(1:6, 1500, replace = TRUE)
```

¿Cuántos elementos del vector `dado2` *son iguales a 4? Responde sin usar* `table`. *Solución en la página 53.* □

7. Formato del código R en próximos tutoriales.

En este tutorial hemos tratado de presentar el código en R de la forma que creemos resulta más natural para los usuarios que acaban de iniciarse en el manejo del sistema. Esa presentación tiene, como decimos, la ventaja de la naturalidad, pero no es la más cómoda para el trabajo que vamos a hacer en el resto del curso. En los próximos tutoriales vamos a utilizar un sistema diferente, pensado para hacer tu trabajo más cómodo. No te lo hemos presentado del principio porque creemos que, sin conocer un poco el funcionamiento de R, podría crear confusión.
¿De qué se trata? Vamos a usar como ejemplo una simulación del lanzamiento de un dado 1000 veces, en la que calcularemos además la media, el rango intercuartílico y la cuasidesviación típica del vector de resultados. Hasta ahora, para mostrarte el código necesario, y los resultados que produce, hemos usado un formato como este.

```
> set.seed(2014)
> dado = sample(1:6, 1000, replace=TRUE)
> mean(dado)
[1] 3.607
> IQR(dado)
[1] 3
> sd(dado)
[1] 1.674322
```

Ese formato está bien, en tanto que reproduce de una forma fiel el "diálogo" que mantenemos con R. Pero si quieres copiar ese código desde este documento, para pegarlo en RStudio, por ejemplo, y así poder ejecutarlo o modificarlo a tu gusto, entonces aparecen algunos inconvenientes. Para empezar, al copiar las líneas de comandos estarás copiando también el símbolo `>` (el *prompt* de R). Y eso significa que tienes que eliminar ese símbolo antes de ejecutar la línea de código. Además, estarás copiando también las líneas de resultados, que no puedes ejecutar sin que se produzcan errores. En resumen, no es un formato cómodo, y lo va a ser cada vez menos a medida que avancemos en el curso. Para evitar esos problemas, vamos a utilizar este otro formato:

```
> set.seed(2014)
> dado = sample(1:6, 1000, replace=TRUE)
> mean(dado)
```

```
[1] 3.607

> IQR(dado)

[1] 3

> sd(dado)

[1] 1.674322
```

Como ves, aparte de los colores y el sombreado (que no son importantes), en este formato ha desaparecido el *prompt* `>` de R. Así que puedes copiar las líneas de código completas en RStudio y ejecutarlas directamente sin problemas.Además, para facilitar la copia de bloques enteros de líneas de código, las líneas con respuestas de R (líneas de salida) se muestran comentadas (precedidas por `#`). De esa manera, aunque las copies a RStudio no causarán ningún efecto sobre la ejecución de los comandos. Este formato para el código y las respuestas simplificará mucho nuestro trabajo en el resto del curso. Y más adelante, en un tutorial opcional explicaremos en detalle la herramienta que hemos usado, porque su utilidad va mucho más allá de un simple cambio de formato.

8. Ejercicios adicionales y soluciones.

Ejercicios adicionales.

28. En todos los ejemplos de aritmética vectorial que hemos visto, los dos vectores implicados eran de la misma longitud. ¿Qué sucede si no es así? Pues algo interesante, y que en cualquier caso conviene conocer para evitar errores. Te invitamos a que lo descubras ejecutando este ejemplo:

    ```
    vector3=c(1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20)
    vector4=c(0,1,2)
    vector3+vector4
    ```

    ¿Qué ha hecho R? Puedes entenderlo así: al principio empieza sumando normalmente, posición a posición. Pero tras sumar tres posiciones, R descubre que `vector4` se ha agotado. Y lo que hace es simplemente volver al principio de `vector4` y comenzar de nuevo a utilizar sus elementos desde el primero. Así pues, en el resultado, se suma el vector 0,1,2 con 1,2,3 y luego se vuelve a sumar 0,1,2 con 4,5,6, (para dar 4,6,8 de resultado) y así, consecutivamente, vamos recorriendo la suma hasta que el vector más largo, `vector3`, se agota a su vez.

    Como ejercicio para practicar esto: dado el vector `vector1 = 68:213`, súmale un `vector2`, de longitud 5, de manera que todos los elementos de `vector1 + vector2` sean múltiplos de 5.

29. En el Tutorial-01 hemos visto cómo calcular la media, varianza, etc. a partir de una tabla de frecuencias de una variable cuantitativa discreta (recuerda, para empezar, la Sección **??**, pág. **??** de ese tutorial). Aquí no podemos, todavía, trabajar con los mismos ficheros que usábamos en aquel caso, porque aún no hemos visto como leer desde R ficheros `csv` que contienen tablas (con varias columnas). Pero podemos trabajar a partir de dos vectores, uno con los valores y otro con las frecuencias, como hemos hecho en la Sección 6.2 (pág. 40).

    a) Usando los vectores que vimos en esa sección:

    ```
    valores = c(2,3,5,8,13)
    frecuencias = c(5,7,12,2,14)
    ```

    calcula la media aritmética, la varianza y cuasivarianza y la mediana. ¡No uses todavía la función `rep`, lo harás en el tercer apartado!!

    b) Calcula también las tablas de frecuencias relativas, acumuladas y relativas acumuladas.

    c) Ahora, usa la función `rep` para fabricar un vector que contenga los mismos datos sin agrupar por frecuencias. Y repite los cálculos de los dos anteriores apartados para comprobar los resultados.

Soluciones de algunos ejercicios.

- **Ejercicio 2, pág. 9**

Al final las variables valen $a = 15$, $b = 100$, $c = 1500$.

- **Ejercicio 3, pág. 15**

```
(df.Edades = read.table(file = "../datos/Tut02-Edades.csv"))
(vectorEdades <- df.Edades$V1)
```

- **Ejercicio 4, pág. 15**

```
(tabla1 = read.table(file = "../datos/Tut02-tabla1.csv", sep =";", header = TRUE))
(vectorSex1 <- tabla1$sex1)
(vectorAge1 <- tabla1$age1)
(vectorSystbp1 <- tabla1$sysbp1)
class(vectorSex1)
class(vectorAge1)
class(vectorSysbp1)
```

- **Ejercicio 5, pág. 15**

```
> errorDeLectura = read.table(file = './datos/EsteFicheroNoExiste.csv')
Error in file(file, "r") : cannot open the connection
In addition: Warning message:
In file(file, "r") :
  cannot open file './datos/EsteFicheroNoExiste.csv': No such file or directory
```

- **Ejercicio 7, pág. 18**

Para guardar el `vector1`, usamos:

```
vector1 = c(8, 5, 19, 9, 17, 2, 28, 18, 3, 4, 19, 1)
```

Para `vector2`, asegúrate primero de que el fichero `Tut02-vector2.csv` está en la subcarpeta `datos` de tu directorio de trabajo, y luego ejecuta:

```
> (df.vector2 = read.table(file="../datos/Tut02-vector2.csv",sep=";") )
(vector2 <- df.vector2\$V1)
 18 18 20 12 20  1 30  6 26 20 15 25
```

- **Ejercicio 8, pág. 20**

```
> (cuadrados = (1:10)^2 )
 [1]   1   4   9  16  25  36  49  64  81 100
>
> (cuadrados2 = (2:11)^2 )
 [1]   4   9  16  25  36  49  64  81 100 121
>
> cuadrados2 - cuadrados
 [1]  3  5  7  9 11 13 15 17 19 21
```

La diferencia entre dos cuadrados consecutivos n^2 y $(n+1)^2$ es siempre el número impar $2n+1$.

- **Ejercicio 9, pág. 21**

```
> sum((1:10)^2)
[1] 385
> # No confundir con
> sum(1:10)^2
[1] 3025
```

• **Ejercicio 10, pág. 21**

```
> df.Edades = read.table(file="../datos/Tut02-Edades.csv")
  vectorEdades <- df.Edades\$V1
> ( n = length(vectorEdades) )
[1] 100
> ( mediaEdades = sum(vectorEdades) / n )
[1] 6.75
> ( varEdades = sum( ( vectorEdades - mediaEdades )^2) / n )
[1] 1.9875
> ( desvTipicaEdades = sqrt(varEdades))
[1] 1.409787
> ( cuasiVarEdades = sum( ( vectorEdades - mediaEdades )^2) / (n - 1) )
[1] 2.007576
> ( cuasiDesvTipicaEdades = sqrt(cuasiVarEdades))
[1] 1.416889
>
> sum( vectorEdades - mediaEdades )^2
[1] 0
```

Este valor 0 se explica en la Ecuación 2.4 (pág. 35 del libro).

• **Ejercicio 11, pág. 22**
¡Después de establecer el directorio de trabajo! Si no, no funcionará.

```
> df.var3 = read.table("../datos/Tut02-var3.csv")
 var3 <- df.var3\$V1
 length(var3)
```

• **Ejercicio 12, pág. 23**

```
> ( tablaFrecAbs = table(var3) )
var3
  0   1   2   3   4   5   6   7   8   9  10  11  12  13  14  16
  9  25 100 188 244 246 186 131  87  43  28   6   2   2   2   1
> length(tablaFrecAbs)
[1] 16
> sum(tablaFrecAbs)
[1] 1300
> min(tablaFrecAbs)
[1] 1
> max(tablaFrecAbs)
[1] 246
> range(tablaFrecAbs)
[1]   1 246
```

• **Ejercicio 13, pág. 23**

```
> ( n = length( var3 ) )
[1] 1300
> ( tablaFrecRel = tablaFrecAbs / n )
var3
           0            1            2            3            4            5
0.0069230769 0.0192307692 0.0769230769 0.1446153846 0.1876923077 0.1892307692
           6            7            8            9           10           11
0.1430769231 0.1007692308 0.0669230769 0.0330769231 0.0215384615 0.0046153846
          12           13           14           16
0.0015384615 0.0015384615 0.0015384615 0.0007692308
> sum(tablaFrecRel)
[1] 1
```

• **Ejercicio 14, pág. 24**

Para la tabla de frecuencias acumuladas:

```
> ( tablaFrecAcu = cumsum(tablaFrecAbs)  )
 0    1    2    3    4    5    6    7    8    9   10   11   12   13   14   16
 9   34  134  322  566  812  998 1129 1216 1259 1287 1293 1295 1297 1299 1300
```

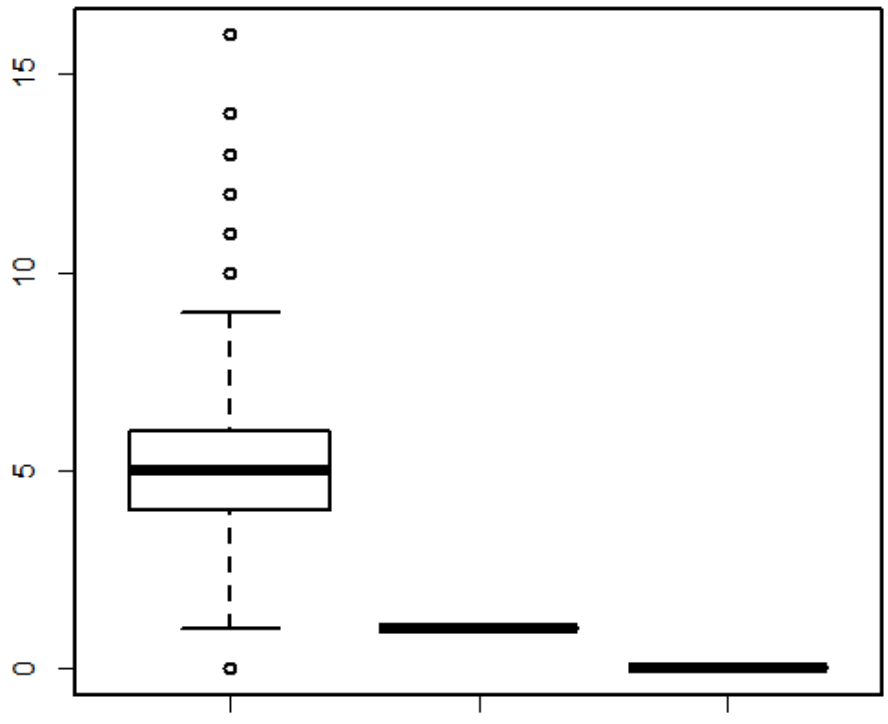
Las dos formas para la tabla de frecuencias relativas acumuladas son:

```
> ( tablaFrecRelAcu = cumsum(tablaFrecRel) )
          0           1           2           3           4           5           6
0.006923077 0.026153846 0.103076923 0.247692308 0.435384615 0.624615385 0.767692308
          7           8           9          10          11          12          13
0.868461538 0.935384615 0.968461538 0.990000000 0.994615385 0.996153846 0.997692308
         14          16
0.999230769 1.000000000
> ( tablaFrecRelAcu = (tablaFrecAcu / n) )
          0           1           2           3           4           5           6
0.006923077 0.026153846 0.103076923 0.247692308 0.435384615 0.624615385 0.767692308
          7           8           9          10          11          12          13
0.868461538 0.935384615 0.968461538 0.990000000 0.994615385 0.996153846 0.997692308
         14          16
0.999230769 1.000000000
```

y, en este caso, producen el mismo resultado. En general, es más recomendable el segundo método.

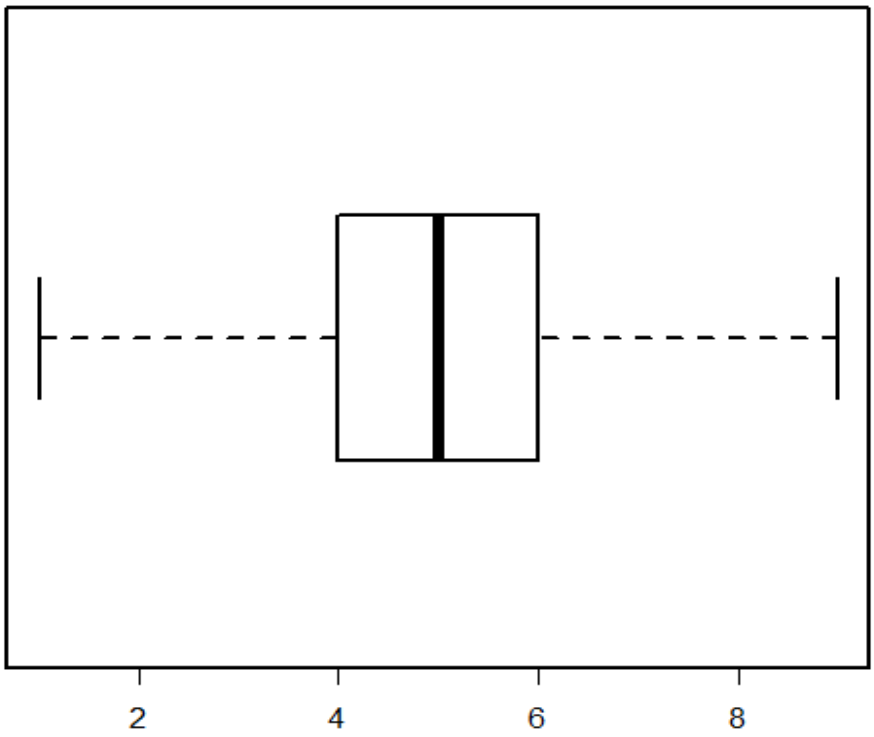
• **Ejercicio 15, pág. 26**

El gráfico resultante será parecido a este:



• **Ejercicio 16, pág. 26**

En ambos casos, el gráfico resultante será parecido a este:



• **Ejercicio 17, pág. 27**
Para la primera parte, después de comprobar que el directorio de trabajo es correcto, hacemos:

```
> df.Edades = read.table(file="../datos/Tut02-Edades.csv")
> (mediaEdades = mean(df.Edades\$V1))
[1] 6.75
```

Para la segunda, las dos posibles formas de calcular la media de `var3` son estas:

```
> ( mediaVar3 = mean(var3))
[1] 5.039231
> ( mediaVar3 = sum(var3)/length(var3))
[1] 5.039231
```

Y, como era de esperar, el resultado es el mismo.

• **Ejercicio 18, pág. 28**
Calculamos los dos valores:

```
> ( varMuestral = var(var3))
[1] 4.715165
> ( desvTipMuestral = sd(var3))
[1] 2.171443
```

Y para la comprobación que pide el ejercicio hacemos:

```
> sqrt(varMuestral)
[1] 2.171443
```

• **Ejercicio 19, pág. 28**
Siempre podemos averiguar de que tipo de objeto se trata con la función `class` de R:

```
> class(summary(var3))
[1] "summaryDefault" "table"
```

En este caso, R nos contesta que es un tipo especial de tabla, asociado precisamente a la función `summary`.

• **Ejercicio 23, pág. 37**

La tabla de frecuencia se obtiene con:

```
> table(dado)
dado
  1   2   3   4   5   6
167 171 167 163 188 144
```

Al ser números aleatorios, tus resultados serán distintos.
Para comprobar que `dado` es un vector, ejecutamos:

```
> class(dado)
[1] "integer"
```

y R nos informa de que, en efecto, es un vector de números enteros (*integer numbers*, en inglés).

• **Ejercicio 24, pág. 39**
El generador de claves modificado es:

```
> ( Clave = sample(c(LETTERS, letters, 0:9), size=15, replace=TRUE) )
 [1] "S" "F" "3" "U" "z" "i" "g" "L" "4" "3" "M" "S" "s" "u" "y"
```

Y como aperitivo de lo que ofrece la función `paste`:

```
> paste(Clave, collapse="")
[1] "SF3UzigL43MSsuy"
```

• **Ejercicio 25, pág. 42**
El comando:

```
> tail(dado, 1)
[1] 1
```

muestra que el último elemento es un 1. Para ver los 10 últimos elementos:

```
> tail(dado, 10)
 [1] 4 6 3 4 3 5 4 2 3 1
```

• **Ejercicio 26, pág. 44**
Apartado 1:

```
> dado[16]
[1] 3
```

Apartado 2. El operador adecuado es `>=`, y usamos `head` para ver el comienzo de la solución:

```
> masoIgualQue2 = dado[dado>=2]
> head(masoIgualQue2)
[1] 2 2 4 2 4 6
```

Apartado 3:

```
> pares = dado[dado%%2==0]
> head(pares)
[1] 2 2 4 2 4 6
```

• **Ejercicio 27, pág. 45**

```
> (7 < 5) | (2 < 4)
[1] TRUE
```

• **Ejercicio 28, pág. 46**

Recuerda que los valores booleanos se traducen en unos y ceros, y escribe:

```
> sum(dado2 == 4)
[1] 280
```

Fin del Tutorial-02. ¡Gracias por la atención!