Introducción al análisis de datos con R

Ajuntament de Barcelona
Barcelona Activa

Índice

1 INTRODUCCIÓN A R Y AL ANÁLISIS DE DATOS

En este primer módulo explicaremos el proceso de instalación de R y su interfaz gráfica RStudio.

En segundo lugar, presentaremos R como lenguaje de programación, haciendo especial énfasis en la utilidad práctica que podemos darle en nuestros procesos de análisis de datos.

Posteriormente, expondremos las principales opciones que nos ofrece esta interfaz gráfica y exploraremos las maneras más sencillas de sacarle el mayor potencial posible. Aprenderemos a crear y guardar un script, a aprovechar la cuadrícula de RStudio para gestionar la información disponible y hacer un uso eficiente de sus funcionalidades.

1.1 QUÉ ES R Y PARA QUÉ SIRVE

¡Hola a todos y todas!

R es un lenguaje de programación de licencia abierta y totalmente gratuito para la computación estadística y la creación de gráficos. Recientemente ha ido incrementando su popularidad en el ámbito de la ciencia de datos, y junto con Python y Java ocupa las primeras posiciones de este sector.

Como muchos otros lenguajes de programación, se fundamenta en la escritura en línea de comandos. Aún así, está extremadamente extendido el uso de la interfaz gráfica RStudio, por su practicidad y estructura. R también destaca por tener una comunidad muy activa que desarrolla paquetes, especialmente orientados a la modelización, la estadística más clásica y la visualización de datos, que son los puntos fuertes del lenguaje. R dispone de un repositorio de paquetes extensísimo, donde podemos encontrar grupos de prácticamente todas las variantes de la ciencia de datos.

A continuación, haremos una lista de algunas de las principales particularidades de R, que deberíamos conocer como usuarios y usuarias:

1. R es lento. Esto puede parecer curioso como primera característica, pero tiene una justificación. R fue diseñado para hacer fácil al usuario o usuaria la modelización y el análisis estadístico, no por ser el lenguaje más amable para un ordenador. Debemos ser conscientes de que no puede competir en velocidad con otros lenguajes, pero sí ganar en usabilidad.
2. R es extremadamente popular en algunos sectores, y en cambio, prácticamente desconocido en otros. Está prácticamente desaparecido en un entorno más orientado a la ingeniería o la informática, pero domina en la industria farmacéutica, las finanzas, el marketing, los audiovisuales y sobre todo la academia, ya que es donde más se usa la estadística formal, uno de los puntos más fuertes de R. Su importancia como herramienta de *Business Intelligence* es cada vez mayor, principalmente por la

simplicidad de su código y las facilidades que ofrece a la hora de hacer visualizaciones y resúmenes estadísticos.

3. Los modelos en R son increíblemente fáciles de usar. Como R es un lenguaje que prioriza la usabilidad, la modelización y su código asociado son mucho más sencillos que en otros lenguajes y resultan más cercanos para personas no expertas en *data science*.
4. R funciona mucho más rápidamente si utilizamos un tipo de código orientado al uso de estructuras vectoriales y planificamos correctamente las variables que necesitaremos. A diferencia de Python, por ejemplo, algunos bucles y especialmente la concatenación de variables pueden ser muy ineficientes. Pero, como ya hemos comentado, una buena planificación y el uso de alternativas que aprovechen la estructura vectorial del lenguaje para ejecutarse rápidamente pueden ayudar a compensar el punto anterior, que es su baja velocidad.
5. R es más complicado de aprender al principio. Sin embargo, cuando ya hayamos entrado en su lógica y ya hayamos realizado un par de proyectos, su código nos parecerá muy intuitivo y aprenderemos nuevas funcionalidades mucho más rápidamente. Los primeros pasos programando con R son más complejos que otros lenguajes más humanos como Python, donde cada tipo de objeto tiene unas funciones asociadas. En R, estas funciones, por ejemplo, las asociadas a un tipo concreto de modelo, tendremos que conocerlas de antemano o recurrir a la ayuda que nos ofrece el programa, lo que ralentiza el proceso de escritura del código al principio. Afortunadamente, la ayuda que nos ofrece R es muy completa y llena de ejemplos, y en la gran mayoría de paquetes incluyen buenos tutoriales.

En resumen, R es un lenguaje de programación orientado al análisis estadístico y a la visualización de datos. Puede ser más complicado de aprender, al principio, que otros lenguajes, pero, debido a su creciente popularidad y versatilidad, su dominio constituye una competencia de gran valor en el mercado laboral.

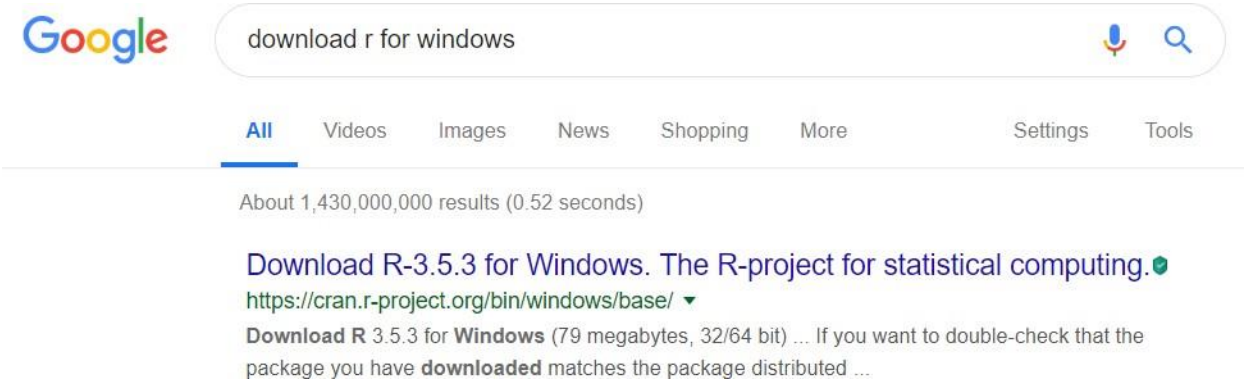
¡Seguimos!

1.2 INSTALACIÓN DE R

¡Hola de nuevo!

En este artículo vamos a avanzar en el proceso de instalación de R, veremos que es muy sencillo, independientemente del sistema operativo con el que estemos trabajando.

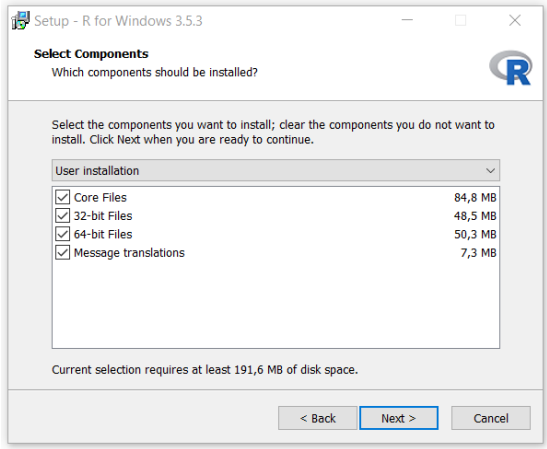
El primer paso que realizaremos es buscar "Install R" en Google y entraremos en la primera página que nos aparezca.

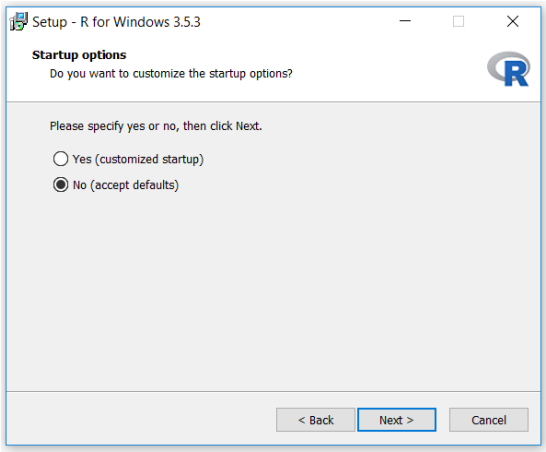


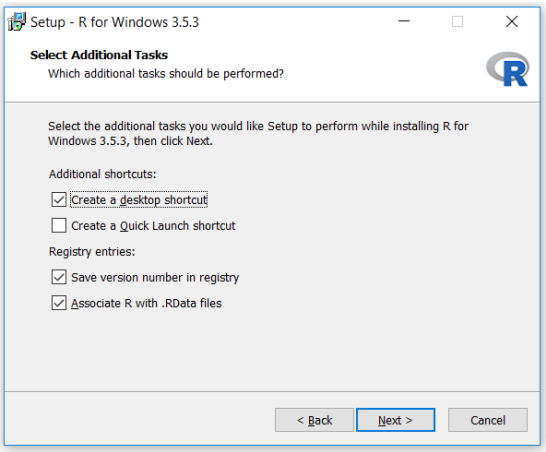
Quizás la web no es suficiente amable, ya que se trata de un proyecto libre. Tendremos que hacer clic en "Download R", resaltado en negrita, y nos redirigirá a una página de servidores.

En principio, debería ser indiferente cuál de ellos seleccionamos, así que hacemos clic con el primero. A continuación deberemos seleccionar nuestro sistema operativo. En este caso, utilizaremos Windows para este tutorial, aunque los pasos son prácticamente idénticos para todos los sistemas operativos.

En la página siguiente, volveremos a elegir la opción resaltada en negrita "Install R for the first time". El siguiente paso es el definitivo y descargaremos el archivo de instalación, que podemos ejecutar para que se nos abra el asistente de instalación.







¡Hasta pronto!

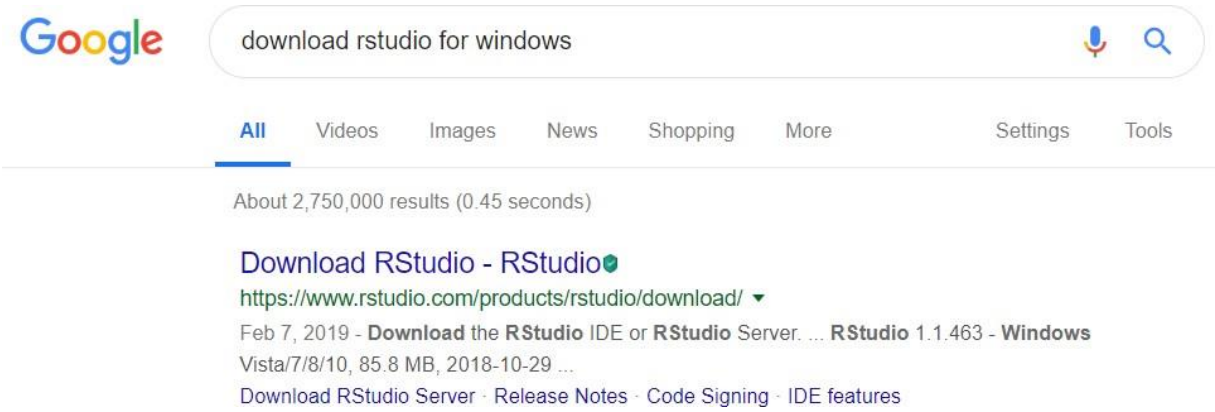
1.3 INSTALACIÓN DE RSTUDIO

¡Hola de nuevo!

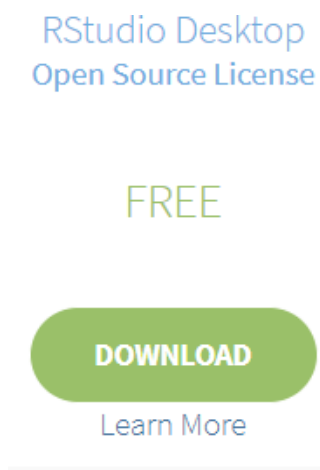
En este artículo vamos a avanzar en el proceso de instalación de RStudio, una interfaz gráfica que nos facilitará enormemente el trabajo, a la vez que nos permitirá una visualización más

agradable y estructurada del entorno de programación. Como es de imaginar, debemos tener completamente instalada la última versión disponible de R para llevar a cabo este proceso.

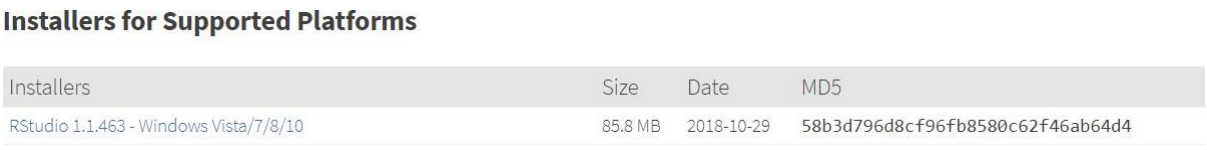
El primer paso que realizaremos será buscar "Install RStudio" en Google y entraremos en la primera página que nos aparezca.



Aquí nos ofrecerá varias opciones; nosotros escogeremos la primera de todas, que es gratuita y completamente funcional.



Cuando pulsamos el botón "Download", nos llevará más abajo en la misma página, donde podremos elegir nuestro sistema operativo. Es tan sencillo como pulsar en la primera opción, en el caso de Windows.



Se nos descargará el archivo que abrirá el instalador. Habrá que ir haciendo clic en siguiente hasta que se complete la instalación.

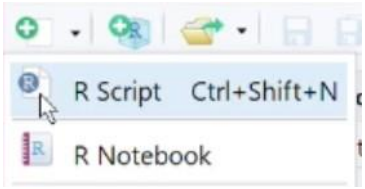
¡Continuamos con el curso!

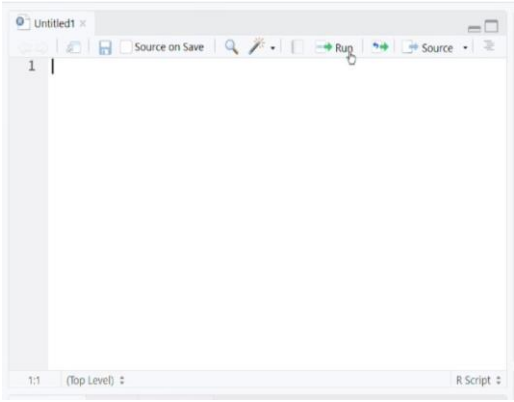
1.4 INTRODUCCIÓN AL ENTORNO DE PROGRAMACIÓN RSTUDIO

¡Bienvenidos y bienvenidas!

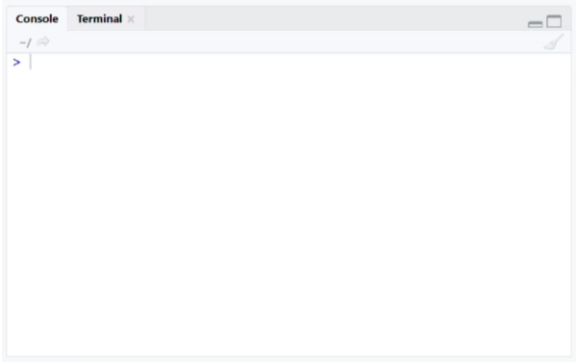
En este vídeo presentaremos la interfaz gráfica RStudio, así como las funcionalidades más interesantes que nos puede ofrecer. Antes de comentar un poco más en detalle lo que nos encontraremos, debemos tener en cuenta un consejo: Si tenemos un cierto dominio del inglés, lo más práctico es ver la interfaz en este idioma. De este modo, en el caso de necesitar ayuda en algún momento, será mucho más sencillo encontrarla a través de Google y aplicarla.

La estructura de RStudio se basa en una parrilla. Es decir, en cuatro elementos. Ahora mismo, sólo vemos tres. Lo que pasa es que todavía no hemos creado un script. El script lo podemos crear utilizando este icono aquí:

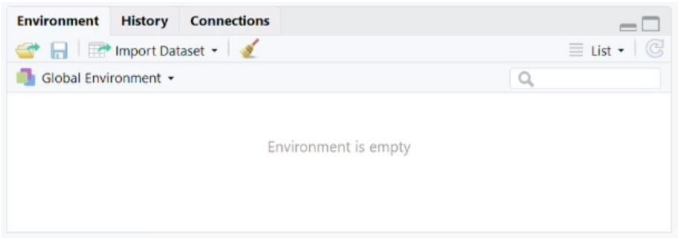


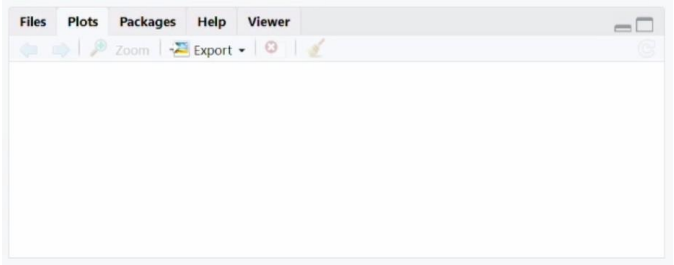


Y abriendo el documento donde escribiremos nuestro código. Todo lo que escribimos aquí y ejecutamos utilizando este botón aquí, lo veremos por la consola.

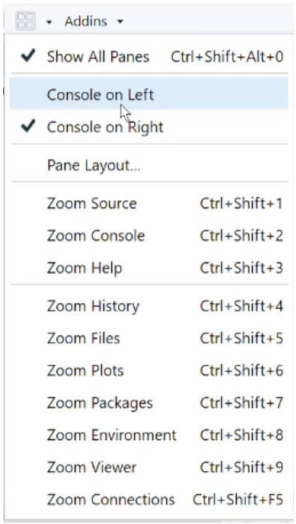


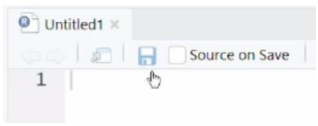
Aquí, alrededor, veremos los objetos que hemos creado y, por ejemplo, los gráficos que estemos generando.



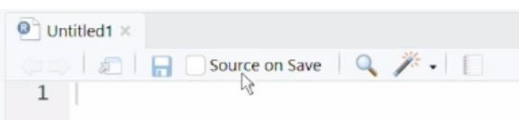


Los elementos más importantes son estos dos de aquí arriba. Utilizando este botón aquí, podemos mover la consola de banda, por ejemplo. Lo hemos puesto a la izquierda y ahora tendríamos un espacio ajustable, donde podemos ver nuestro script, nuestra consola y, en otra parte, podríamos ver nuestro entorno. Guardar un script es tan sencillo como utilizar este botón de aquí.





Otra función muy importante que necesitamos saber si trabajamos con RStudio es esta aquí: el Working Directory.



Es decir, dónde queremos que se nos guarden los archivos. Si hacemos clic aquí, lo que estamos consiguiendo es que todos los gráficos que generamos y objetos nuevos que queramos guardar se guarden exactamente en la misma carpeta donde tenemos el script que acabamos de generar. También podemos escogerlo utilizando esta opción aquí. Aquí arriba tenemos una gran cantidad de menús que permiten personalizar como vemos nuestro RStudio. Tenemos las opciones globales, donde podemos, por ejemplo, ajustar la visualización de nuestro R.

Estos ajustes son puramente estéticos. Lo que es especialmente útil a la hora de utilizar RStudio, que no nos ofrece R, es esta visualización en formato parrilla, donde podemos ver nuestros gráficos, los paquetes que queramos cargar, ayuda, un entorno donde veremos las variables, un histórico del que hayamos hecho, una consola, el script y tendremos ayuda en el caso de necesitarla.

¡Seguimos!

1.5 IDEAS CLAVE: INTRODUCCIÓN A R I AL ANÁLISIS DE DATOS

En este módulo hemos realizado los primeros pasos con R y RStudio. A continuación, haremos un repaso de las ideas más importantes que han aparecido:

- R es un conocido lenguaje de programación que ha crecido enormemente en los últimos años. Se caracteriza por estar totalmente orientado a la usabilidad, modelización y visualización, y está ampliamente extendido en muchos sectores.
- Hemos instalado R y su interfaz gráfica RStudio, a la vez que hemos explorado las funcionalidades principales de RStudio, como por ejemplo:
  - Visualización de los objetos creados
  - Estructura y legibilidad del código
  - Asistentes de importación
  - Gestión de scripts y de proyectos

2 PROGRAMACIÓN CON R

En este módulo veremos un resumen muy breve de los fundamentos de R como lenguaje de programación.

Comenzaremos presentando los cuatro grandes tipos de objetos con los que podemos trabajar, cuáles son sus particularidades y para qué son útiles.

A continuación, veremos cómo funcionan las estructuras condicionales más sencillas y algunas más complejas, lo mismo con bucles de diferentes tipos, haciendo especial énfasis en sus condiciones de parada.

A lo largo de este módulo veremos, de manera transversal, algunas de las funciones básicas de R.

2.1 DIFERENTES TIPOS DE OBJETOS EN R (I)

¡Hola de nuevo!

En este vídeo veremos los diferentes tipos de objeto que podemos crear en R, cuál es su utilidad principal y cómo podemos tratarlos adecuadamente. Veremos cómo crear variables unitarias, vectores y matrices.

Lo primero que haremos será crear una variable unitaria. Le asignamos un nombre y, utilizando esta flecha, le asignaremos un número.

```
num <- 3
```

Para ejecutar esto que acabo de escribir, puedo apretar Run o puedo pulsar Control + Enter. Lo que podemos ver es que nos aparece en nuestro entorno esta variable que acabamos de crear y la instrucción que hemos ejecutado en nuestra consola.

```
num <- 3.0
```

Otra cosa que podemos hacer es crear un texto. Toda la creación sigue la misma estructura: un nombre y una flecha. Le pongo estas cuatro letras, ejecuto y ahora he creado una variable que se llama "texto" y que contiene estas cuatro letras.

```
texto <- "asdf"
```

Otro tipo de estructura es el vector. Le pondré el nombre "numeros" y con la flecha y esta instrucción aquí: c, abro paréntesis y le puedo añadir los números que quiera.

```
numeros <- c (1,2,3)
```

Ejecuto y veo que aquí abajo me dice el nombre, el tipo de variable que es (es decir, numérica), y que tiene tres posiciones, 1, 2 y 3.
Una manera alternativa de crear este mismo objeto podría ser esta de aquí.

```
numeros <- 1: 5
```

Hagámoslo del 1 hasta el 5. Ejecuto y lo que veo es que me ha creado un objeto con una secuencia del 1 hasta el 5. Una consideración que hay que hacer sobre los vectores es que todos los elementos que lo componen deben ser del mismo tipo. Me explico: si yo creo este

objeto, pongo 1, 2, 3 y añado un 4 pero en formato texto, que es lo que delimitan estas comillas aquí.

numeros <- c (1,2,3, "4")

Ejecuto. Lo que habrá pasado es que he creado un objeto con cuatro posiciones donde todas ellas son variables textuales, es decir, el tipo más flexible de variable del texto.
Obtiene este nombre de variable.

vector_variado <- c (5, TRUE, "Hola")

Los espacios no están permitidos, así que añado esta barra baja. Puedo poner un número, una variable binaria y un texto. Si ejecuto esta instrucción, creo un nuevo vector donde todo es formato texto, y tengo un 5 en formato texto, una variable binaria en formato texto y un texto.
El último que veremos en este vídeo es cómo crear una matriz. Le pondré de nombre "matriz" y la crearé con la instrucción *matrix*, del 1 hasta el 12, y le puedo especificar el número de columnas que quiero que tenga. Le estoy diciendo: "ponme los números del 1 hasta el 12 en tres columnas".

matriz <- matrix (1: 12, ncol = 3)

Ejecutamos y visualizamos el que acabamos de crear. Ejecuto y tengo esta estructura aquí: tres columnas, cuatro filas y los números me ha llenado en vertical. Puedo acceder a los elementos de esta matriz utilizando los corchetes. Por ejemplo, puedo coger el objeto que ocupa la primera fila y la segunda columna.

matriz [1,2]

En la primera fila y la segunda columna será el número 5.
Lo mismo podríamos hacer diciéndole que nos coja la primera fila y la segunda y tercera columna.

matriz [1,2: 3]

También podemos omitir uno de estos dos elementos y, si lo dejamos en blanco, lo que estaríamos haciendo es elegir sólo la primera fila.

matriz [1,]

2.2 DIFERENTES TIPOS DE OBJETOS EN R (II)

¡Hola!

En este segundo vídeo exploraremos cómo crear y manipular dataframes y listas, los objetos más versátiles y utilizados en R.

Lo primero que haremos es cargar el paquete datasets.

```
require(datasets)
```

Lo que obtendremos dentro de este paquete son un conjunto de bases de datos de ejemplo. Utilizaremos esta, que consiste en un conjunto de coches con sus características.

```
mtcars
```

Guardaremos esta base de datos como df, que es la notación clásica para dataframe.

```
df <-mtcars
```

El ejecutamos y lo que podemos ver es como accedemos a la información de una de las columnas. La sintaxis que se utiliza es el $ y nos aparece una lista con las columnas que tiene este objeto. Si ejecuto esta instrucción, puedo obtener un vector con cada uno de los elementos de la columna de cilindros para cada uno de estos coches.

```
df $ CYL
```

Una forma alternativa de hacerlo es utilizando los corchetes.

```
df [ "CYL"]
```

El output que obtenemos es ligeramente diferente, ya que aquí nos aparecen los nombres de las filas, lo que es más práctico. Si quiero obtener varias columnas a la vez, puedo utilizar esta sintaxis aquí.

```
df [1: 3]
```

Si, en cambio, lo que hago es añadir una coma (,) y pongo un nombre de columna, como si fuera un texto, ahora lo que obtengo son los valores para las tres primeras filas de la columna "cilindros" (CYL).

```
df [1: 3, "CYL"]
```

Esta misma estructura la puedo utilizar en formato vectorial y lo que puedo conseguir es seleccionar más de una columna a la vez. Por ejemplo seleccionaremos la columna "mpg", tres filas y las dos columnas que le hemos pedido.

```
df [1: 3, c ( "CYL", "mpg")]
```

Vamos a ver otro tipo de estructura, que es la lista. Una lista se crea utilizando esta función.

```
lista <- list ()
```

Si la creamos vacía, tenemos una lista de cero elementos. Para añadir elementos a una lista, podemos utilizar también el $. Por ejemplo, vamos a crear un primer elemento de esta lista y le diremos "objeto1", y aquí guardaremos un vector con tres números.

```
lista $ objeto1 <- c (1,2,4)
```

Ahora tenemos una lista con un elemento, no con tres, ya que en esta lista el primer elemento es un vector. Lo que puedo hacer es crear otro. Aquí guardaremos "Esto es un texto" y tenemos una lista de dos elementos.

```
lista $ objeto2 <- "Esto es un texto"
```

Como puedes comprobar, es un formato muy flexible, ya que puedo guardar un vector, un texto y también podemos guardar, por ejemplo, una fracción de nuestro dataframe. Por ejemplo, las tres primeras filas.

```
lista $ objecte3 <- df [1: 3,]
```

Para acceder a la información que hemos guardado, podemos visualizarla así y veríamos el primer elemento, un vector; el segundo elemento, un texto; y el tercer elemento, una base de datos. Podemos acceder o por los nombres, como hacíamos con los dataframes, o utilizando los dobles corchetes.

```
lista [[1]]
```

Podríamos obtener el primer elemento así y, de este primer elemento, podríamos obtener la segunda posición, utilizando esta sintaxis aquí.

```
lista [[1]] [2]
```

Así pues, si quisiéramos explorar el dataframe que hemos guardado aquí, deberíamos hacer "lista", "$", "objeto3"

lista $ objeto3

que sería esta estructura aquí. Y ahora podríamos hacer, por ejemplo, selecciona'm la columna "CYL".

lista $ objeto3 [, "CYL"]

Y obtendríamos estos tres valores de aquí en formato vectorial.

2.3 CONDICIONALES

¡Hola a todos y todas!

A continuación, veremos qué son las condiciones en un lenguaje de programación y cómo podemos crear nuestras propias con R.

Una condición es aquella estructura de control que utilizamos para decidir, basándose en un cierto criterio, si queremos realizar alguna acción o no. La manera más sencilla de ver cómo funcionan las condiciones es con un ejemplo:

Lo primero que haremos es crear una nueva variable que le diremos "edad", que representará nuestra edad.

```
(Edad <- 26)
```

La estructura condicional que utilizaremos es esta de aquí.

```
if (edad> = 18) {
```

Donde, aquí dentro del paréntesis, especificaremos una condición, por ejemplo, si mi edad es mayor o igual a 18 años, y aquí dentro de estos corchetes qué acción queremos realizar.

```
print ( "Mayor de edad")
}
```

Si ejecutamos esta variable y la estructura condicional, obtenemos que el individuo es mayor de edad; si, en cambio, modificamos esta variable, esta estructura no nos devuelve absolutamente nada.

```
Edad <- 16
if (edad> = 18) {
print ( "Mayor de edad")
}
```

Lo que podemos hacer, sin embargo, es mejorar esta estructura, diciéndole que queremos que haga en el caso contrario, "en caso de que esta condición no se cumpla, hazme esto de aquí".

```
else {
print ( "Menor de edad")
}
```

Si ejecutamos toda esta estructura, lo que nos devuelve es que el individuo es menor de edad.

Una manera alternativa de realizar estas estructuras es utilizando la función "ifelse". La función "ifelse" depende de tres parámetros: nuestra condición, que será la edad.

```
ifelse (test = edad> = 18)
```

¿Qué queremos que haga, si se cumple, esto de aquí.

```
ifelse (test = edad> = 18, yes = print ( "Mayor de edad"))
```

Y, ¿qué queremos que haga, cuando no se cumple.

```
ifelse (test = edad> = 18, yes = print ( "Mayor de edad"), no = print ( "Menor de edad"))
```

Si lo ejecutamos, vemos que nos está dando la segunda respuesta.

Esta es una introducción muy breve el funcionamiento de las condiciones con R. Aquí hemos propuesto una estructura basada en una sola condición, que, si se cumplía, imprimía un resultado y, si no se cumplía, imprime otro. Evidentemente, podemos definir condiciones múltiples, como por ejemplo creando una nueva variable que sea nuestra edad.

```
edad <- 20
```

Y si el individuo es hombre o mujer.

```
sexo <- "H".
```

Utilizando la estructura "if", cabe preguntarse si la edad es mayor o igual a 18 años y utilizando "&" el sexo es igual a "Home". Esta estructura de aquí pregunta si se cumple esta condición y esta otra.

```
if (edad> = 18 & sexo == "H")
```

Aquí también hemos introducido la comparación de igualdad. Aquí estábamos haciendo una desigualdad en la que incluíamos igual (> =) y aquí estamos haciendo exactamente una igualdad (==). Queremos que nos diga "Hombre adulto".

```
if (edad> = 18 & sexo == "H") {
print ( "Hombre adulto")
}
```

Así pues, ejecutamos estas variables y, ahora, cuando ejecutamos esta condición múltiple, nos dirá que las cumple ambas, ya que es lo que le estamos preguntando. Por ejemplo, si cambiáramos "Home" en "Mujer", ahora ejecutamos esta condición y ya no se cumple.

```
sexo <- "D".
```

Podemos modificar estas condiciones para que se adapten a todo tipo de circunstancias. Por ejemplo, este símbolo aquí, (|) la raya en vertical, indica una condición u otra condición.

```
if (edad> = 18 | sexo == "H") {
print ( "Hombre adulto")
}
```

Si lo ejecutamos, nos saldrá "Hombre adulto". ¿Por qué? Porque está evaluando si la edad es mayor que 18, que en este caso es verdad, o si es hombre. En este caso no es verdad, pero como ya se cumplía la primera, nos imprime el que tenemos aquí dentro.

Una última cosa es que podemos modificar esta doble igualdad por una desigualdad. Aquí, lo que le estamos diciendo a R es que nos compruebe si el sexo no es igual, ("! =") A hombre.

```
if (edad> = 18 | sexo! = "H") {
print ( "Hombre adulto")
}
```

Y hasta aquí esta introducción a las condiciones. ¡Continuamos!

A continuación os proponemos unos ejercicios donde tendremos que utilizar múltiples veces la condicional () para generar estructuras condicionales más complejas. La dificultad de estas estructuras radica en donde definimos las diferentes condiciones, y en qué orden lo hacemos!

Ejercicios:

1. Crea el código que, facilitando un año de nacimiento concreto, te diga si has nacido antes del 80, entre 1980 y 1999, o a partir del año 2000.

2. La instrucción ifelse () también funciona cuando la aplicamos en vectores. Crea vector c (-5, 4, 8, -1) y, utilizándola, consigue el vector c ("Negativo", "Positivo", "Positivo", "Negativo").
3. Utilizando los paréntesis adecuados, que determinan la importancia de las operaciones, y una sola condición, encuentra una manera de detectar si un número es inferior a 18 o superior a 99, y al mismo tiempo es par. Tendrás que utilizar la siguiente condición: numero %% 2 == 0. Explora el funcionamiento antes de comenzar el ejercicio. Qué hace numero %% 2?

Para resolver los ejercicios, encontraremos la solución a continuación.

2.4 RESOLUCIÓN DE EJERCICIOS: CONDICIONALES

¡Hola de nuevo!

A continuación veremos cómo resolver los ejercicios planteados.

Lo primero que pedíamos era hacer un filtro que clasificara un año en función de otras categorías.

```
Año <- 1993
```

Partiendo de este año, miraremos si es mayor o igual al año 2000. Si nos encontramos en los 80 o 90 o antes de los 80. Lo primero que hacemos es comprobar si este año es mayor o igual que el año 2000. En caso afirmativo, imprimimos esta instrucción.

```
if (año> = 2000) {
print ( "A partir de los 2000")
```

En caso contrario, realizamos toda esta estructura de aquí, que es un condicional en sí.

En caso de que esta condición no se haya cumplido, miraremos si el año es mayor o igual que en 1980. En caso afirmativo, nos encontramos en los 80 y 90.

```
} Else {
if (año> = 1980) {
print ( "80s-90s")
```

En caso contrario, por fuerza, nos encontraremos antes de los 80.

```
} Else {
print ( "Antes de los 80")
}
```

Si ejecutamos toda esta instrucción, obtenemos el resultado deseado.

```
if (año> = 2000) {
print ( "A partir de los 2000")
} Else {
if (año> = 1980) {
print ( "80s-90s")
} Else {
print ( "Antes de los 80")
}
```

A continuación, mostraremos una solución para el segundo ejercicio, que básicamente te pedíamos que utilizando este vector aquí "c (-5,4,8, -1)" clasificaras los números según si son negativos o positivos. Esto es tan sencillo como utilizar la función ifelse aplicando la condición de que, si cada uno de los números es más pequeño que 0, ejecute esta parte de aquí ("Negativo") y, en caso contrario, esta aquí ("Positivo").

```
numeros <-c (-5,4,8, -1)
ifelse (numeros <0, "Negativo", "Positivo")
```

Elijo el vector, ejecuto y obtengo exactamente lo que me interesaba. Este resultado de ahí puedo guardar como un nuevo vector y así tengo los resultados en un objeto independiente.

```
nouvector <- ifelse (numeros <0, "Negativo", "Positivo")
```

Aquí he omitido la parte yes y no, ya que, por posición, la primera siempre ocupa la parte del yes y la segunda la parte del no.

```
nouvector <- ifelse (numeros <0, yes = "Negativo", no = "Positivo")
```

Y, en este último ejercicio, vemos como aplicar una condición múltiple. Lo que necesitamos es crear una variable, que será nuestro número.

```
numero <- 12
```

Y, dentro de la condición, le pedimos dos cosas, en realidad tres. Por un lado, que se cumpla una de estas dos (numero <18 | numero> 99) y por otra parte, que se cumpla esta (numero %% 2 == 0).

```
if ((numero <18 | numero> 99) & numero %% 2 == 0) {
```

Lo primero que hacemos es decirle que mi número sea menor que 18 o mayor que 99. Esto aquí se cumplirá, es decir será true, cuando mi número cumpla o ésta (numero <18) o esta

(numero> 99). Por otra parte, buscaré que el resto de la división de mi número por 2 sea igual a 0. También es cierto.

Si miramos las dos condiciones conjuntamente, debe cumplirse tanto esta (numero <18 | numero> 99) como ésta (numero %% 2 == 0), ya que lo hemos especificado con el símbolo &.

Si ejecutamos toda esta instrucción:

```
numero <- 12
if ((numero <18 | numero> 99) & numero %% 2 == 0) {
print ( "Cumple todas las condiciones")
}
```

Nos imprime por pantalla que el número cumple todas las condiciones.

2.5 FUNCIONES (I)

¡Bienvenidas y bienvenidos de nuevo!
En este vídeo presentaremos qué son y cómo se utilizan las funciones en R. Podemos simplificar la definición de qué es una función con la siguiente frase: "es toda aquella instrucción en R que va seguida de dos paréntesis". Normalmente, las funciones se caracterizan por llevar a cabo un proceso muy determinado sobre alguno de los objetos con los que estamos trabajando, aunque no siempre es así.

Los ejemplos más clásicos de funciones son aquellos que calculan estadísticos sobre un vector. Por ejemplo, si tenemos estas valoraciones de un producto, por ejemplo, y los aplicamos una función.

```
valoraciones <- c (7,8,6,10,5,7,4,6,10)
```

La más famosa de todas es la media.

```
mean(valoraciones)
```

La media, sencillamente, suma todos estos valores y los divide entre la longitud.
Otra función sería, por ejemplo, la suma.

```
sum(valoraciones)
```

Y otra, la longitud.

```
length(valoraciones)
```

Utilizando la combinación de estas dos funciones,

sum(valoraciones) / length(valoraciones)

podríamos obtener exactamente lo mismo que en la función mean.
Otras funciones muy interesantes son, por ejemplo, la varianza;

var(valoraciones)

la desviación estándar, que básicamente es la raíz de la varianza;

sd(valoraciones)

el mínimo;

min(valoraciones)

el máximo;

max(valoraciones)

la mediana, que separa el 50% de las observaciones superiores del 50% de las observaciones inferiores;

median(valoraciones)

y, una de mis preferidas, la función summary,

summary(valoraciones)

que según el objeto sobre el que la aplicamos, obtenemos un resultado u otro. Sobre un vector, nos dará el mínimo, el primer cuartil, la mediana, la media, el tercer cuartil y el máximo.

El primer y el tercer cuartil son los puntos que separan los 25% de las observaciones por debajo y el 25% de las observaciones por encima. Si quisiéramos obtener información más detallada de los cuantiles, podríamos utilizar la función *quantile*.

quantile(valoraciones)

Si lo ejecutamos, nos dará el mínimo, el 25, el 50, el 75 y el 100. Exactamente los mismos que nos estaba dando la función *summary*. Pero podemos especificarle, por ejemplo, qué probabilidatos queremos. Si queremos el cuantiles 40%, lo podemos hacer con esta instrucción.

```
quantile(valoraciones, probs = .4)
```

2.6 FUNCIONES (II)

Hemos empezado explicando cómo aplicar funciones ya existentes en R, ya que con la paciencia suficiente y con la ayuda de R o Google, seguramente seamos capaces de encontrar una función que realice exactamente el cálculo o proceso que queremos en prácticamente todas las situaciones .

Aún así, siempre podremos definir nuestras propias funciones. ¿De qué nos servirá esto? Pues permitirá guardar un código que queramos aplicar muchas veces de una manera muy compacta y definir el tipo de salida o resultado que deseamos.

Veamos un par de ejemplos muy sencillos:

Primero, vamos a crear nuestra propia función para calcular la media. Definimos un nombre de función, le diremos "media", y, utilizando la instrucción function, que dependerá de unos parámetros que pondremos aquí dentro, realizaremos un cálculo.

```
media <- function () {
}
```

El cálculo que realizaremos es lo que os hemos mostrado antes, la suma de X dividido por la longitud. Así pues, mi función media dependerá de un valor X, un parámetro X, y lo que hará es sumar todo y dividirlo por su longitud, exactamente el cálculo de su media.

```
media <- function (x) {
sum (x) / length (x)
}
```

Si ejecuto esta función, digamos que me aparece como una nueva función, la que acabo de definir ahora mismo. Si creo un objeto y le digo "números", del 1 al 5, por ejemplo.

```
numeros <- c (1,2,3,4,5)
```

Puedo ejecutar directamente y me calcula la media de estos números.

```
media (numeros)
```

Esto que acabo de escribir es equivalente a utilizar la instrucción return.

```
media <- function (x) {
return (sum (x) / length (x))
```

```
}
```

Esto aquí, básicamente, lo que especifica es lo que quiero que me devuelva la función. Si no lo especificamos, sencillamente devuelve el último que encuentra, y en este caso era equivalente.

En segundo lugar, utilizando las condiciones que ya hemos visto anteriormente, detectaremos si nos encontramos en un año bisiesto (básicamente si el año es divisible por 4, para simplificar) o no.

Así pues, creamos esta función, que dependerá de un año.

```
anytraspas <- function (año)
```

Y ahora, aquí dentro, vamos a crear una estructura condicional: si el año dividido por 4 da exacto, nos devolverá "El año tiene 366 días".

```
if (año %% 4 == 0) {
return ( "El año tiene 366 días")
En caso contrario, nos devolverá "El año tiene 365 días".
else {
return ( "El año tiene 365 días")}
```

Como puedes comprobar, podemos utilizar la instrucción return tantas veces como queramos. Y siempre termina lo que hace la función. Así pues, si le añadiéramos un return aquí abajo, como que habría encontrado uno de estos dos, este que hubiéramos puesto aquí ya no se ejecutaría.

```
anytraspas <- function (año) {
if (año %% 4 == 0) {
return ( "El año tiene 366 días")
} Else {
return ( "El año tiene 365 días")
}
```

Ejecutamos la función y ahora veremos lo que nos devuelve si le ponemos 2004, por ejemplo, era un año bisiesto.

```
anytraspas  (2004)
```

O en 2006, que no lo era.

```
anytraspas (2006)
```

Al ser una función, podemos guardar este resultado en una variable. "Resultado 2006", por ejemplo.

```
resultat2006 <- anytraspas (2006)
```

Ejecutamos y ahora vemos que se nos ha ejecutado una variable textual que nos dice que el año tiene 365 días.

A continuación, proponemos algunos ejercicios sobre funciones. Encontraremos la solución de cómo hacerlo más adelante:

1. Define una función que imprima en pantalla la diferencia entre el máximo y el mínimo de un vector.
2. Modifica la función anterior para que, en el caso de recibir un solo número en lugar de un vector de número, imprima en pantalla una advertencia. Puedes hacerlo con la función length ().
3. Si definimos una función como multiplicar <- function (a, b) {return (a * b)} y lo llamamos como multiplicar (2,3), obtendremos el número 6. Así, podemos especificar funciones que dependan de más de una variable. Crea una función que dependa de dos números. La función debe devolver el número más grande e imprimir en pantalla si alguno de los números es negativo.

2.7 RESOLUCIÓ DE EXERCICIOS: FUNCIONES

¡Hola de nuevo!

Veamos cómo resolver los ejercicios de funciones.

Lo primero que he hecho es definir un vector y un número, que es el que utilizaremos para ver cómo funcionan nuestras funciones.

```
vector <- c (1,4,5,7,4,2,4,6,8,9,10,2)
numero <- 7
```

La primera que hemos pedido es crear es la función rango, es decir, la diferencia entre el máximo y el mínimo.

```
rango <- function (x) {
max (x) - min (x)
}
```

Aquí, como sencillamente estamos aplicando esta diferencia, no hace falta utilizar la función return. Si ejecutamos esta función y ahora la llamamos sobre el vector, nos dirá la diferencia entre el número mayor, 10, y el número más pequeño, 1.

rango (vector)

Si ejecutamos esta misma función sobre el número, nos da rango 0.

rango (numero)

El segundo ejercicio busca corregir esto, es decir, busca que nuestra función entienda que, si no le estamos entrando un vector, es decir, si la longitud del objeto que entra es igual a 1, es decir, un número, no un vector, nos dirá: "eh, has introducido un número, no un vector". En caso contrario, será exactamente lo que le hemos pedido. Observa que en esta primera condición no devuelve nada. Sencillamente avisa por pantalla.

```
rango <- function (x) {
if (length (x) == 1) {
print ( "Has introducido un número, no un vector")
} Else {
return (max (x) - min (x))
}
}
```

Si ejecutamos esta función, que estamos sobrescribiendo el anterior, ahora podemos probar qué hace cuando ejecutamos el rango del número. Ya no nos da 0, como anteriormente, sino que nos dice: "Has introducido un número, no un vector".

En el último ejercicio te pedía una función que te dijera cuál de los dos números que le dabas es más pequeño que el otro, y que te avisa si al menos uno de los dos es negativo.

```
mesg <- function (numero1, numero2) {
if (numero1 <0 | numero2 <0) {
print ( "Al menos uno de los números es negativo")
}
if (numero1> numero2) {
return (numero1)
}
return (numero2)
}
```

Estas dos condiciones no son excluyentes, es decir, primero lo que haremos es comprobar si uno de los dos números es menor que 0, utilizando esta condición (numero1 <0) o esta otra (numero2 <0); que nos avisará que al menos uno de los dos es negativo y, independientemente de lo que haya pasado hasta ahora, mirará si el número 1 es mayor que el número 2: nos devolverá el número 1, es decir, el mayor; y, en caso contrario, nos devolverá el número 2. Aquí no es necesaria la estructura else, ya que, si utilizamos este return (return

(numero1)), salimos de la función y, si esta condición no se ha cumplido, llegamos aquí (return (numero2)) y devolverá el número 2.

Vamos a ver cómo funciona esta función. El ejecutamos y la aplicamos sobre el número 7 y el -8, por ejemplo.

```
mesg (7, -8)
```

Ejecutamos y nos da dos resultados: por un lado, lo que nos está devolviendo, es decir el número más grande y, por otro lado, nos avisa de que al menos uno de los dos números es negativo.

Si guardáramos el resultado de esta función en un objeto, por ejemplo, le diremos "obj",

```
obj <- mesg (7, -8)
```

si miramos lo que estamos haciendo es mostrar un resultado por pantalla que no se guarda

```
obj <- mesg (7, -8)
obj
```

y, dentro del objeto, hemos guardado el número 7, como podíamos ver en nuestro entorno.

2.8 BUCLES (I)

¡Bienvenidas y bienvenidos de nuevo al vídeo!

A continuación expondremos de manera muy breve que son los bucles y su utilidad práctica, ya que son una de las estructuras más importantes de los lenguajes de programación, junto con los condicionales y los objetos.

Un bucle, básicamente, es un proceso que repetiremos tantas veces como definimos, y que normalmente se utiliza para explorar los diferentes elementos de un objeto, o realizar alguna operación hasta que se satisfaga alguna condición predefinida.

Veamos un ejemplo práctico. Si queremos explorar este vector, esta lista de números del 1 hasta el 20, lo que hará el bucle FOR que acabamos de definir, que es seguramente el más popular de todos, es ir elemento por elemento y aplicar el proceso que hayamos definido en su interior.

```
listanumeros <- 1:20
```

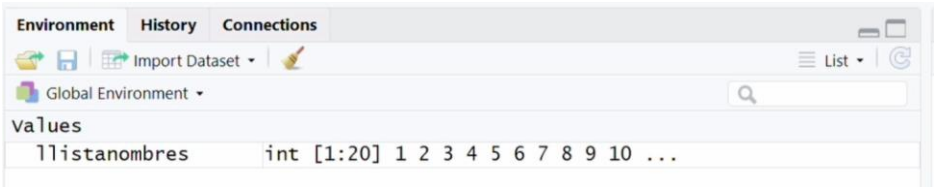
Así pues, defino el bucle con unos paréntesis.

Dentro de los paréntesis, le digo un nombre de una variable, que es arbitrario. Aquí hubiera podido ponerle cualquier nombre que yo hubiera querido y, por cada elemento que encuentre

dentro de este vector o lista, irá llamándole y y aplicará esta operación que le estoy pidiendo. En este caso, le estoy que me imprima cada uno de estos elementos que encuentre en la lista de números elevado al cuadrado. Esta es la sintaxis para elevar un número al cuadrado.

```
for (i in listanumeros) {
print (y ^ 2)
}
```

Así pues, antes de ejecutar, lo que interesa ver es que tomará el primer número, es decir el 1. Este de aquí.



Este número lo guardará como y y la elevará al cuadrado y la imprimirá por pantalla. Llegará aquí y seguirá, ya que aún quedan números en la lista para explorar. El siguiente número es el 2. Guardará con el nombre de y este valor y lo imprimirá elevado al cuadrado. Llegará aquí y, como todavía quedarán números para explorar, cogerá el 3, el 4, etc. hasta llegar al 20, para cada uno de los elementos que haya encontrado en esta lista. Si ejecutamos este vector, vemos que ejecuta 20 veces la instrucción print.

¡Y seguimos!

2.9 BUCLES (II)

Hola de nou!

Otra manera muy clásica de definir los loops o bucles es, en vez de iterar por cada uno de los elementos de una lista, como hemos hecho anteriormente, hacerlo para cada uno de los índices.

```
listanumeros <- c (7,8,9,2,4,5,6,1,7,8,9,10)
for (i in listanumeros) {
print (y ^ 2)
}
```

Así pues, podemos transformar el bucle que teníamos, utilizando no los valores en sí, es decir 7, 8, 9, 2 ... sino la posición que ocupan en el vector, es decir, 1, 2, 3, 4, 5 , etc. Esto es tan sencillo como transformar la lista sobre la que Iter en esta lista de posiciones, del 1 hasta la longitud de nuestro vector.

```
listanumeros <- c (7,8,9,2,4,5,6,1,7,8,9,10)
```

```
for (i in 1: length (listanumeros)) {
print (y ^ 2)
}
```

Si ejecutamos este bucle, obtendremos los números del 1 al 12 elevados al cuadrado.

Pero eso no es lo que nos interesa, sino que nos interesa que el número que ocupa la primera posición del elevamos al cuadrado. Así pues, tenemos que repensar esta instrucción de aquí y transformarla.

```
print (listanumeros [y] ^ 2)
```

De la lista de números, tomaremos, a cada vuelta, la posición "y", y acabamos de conseguir el mismo bucle que teníamos antes.

```
listanumeros <- c (7,8,9,2,4,5,6,1,7,8,9,10)
for (i in 1: length (listanumeros)) {
print (listanumeros [y] ^ 2)
}
```

Finalmente, otra estructura muy útil es el while, que funciona de manera bastante diferente, ya que el criterio de parada es más complejo. Por ejemplo, podemos definir un número:

```
numero <- 1
```

El 1, por ejemplo, y, con esta función aquí, que tendrá una condición de parada aquí dentro, y las acciones que queramos hacer, y luego pondremos que, mientras el número sea menor que 1000, mientras esto se cumpla, nos imprima el número y, cada vez que lo haga, este número se guarde como él mismo multiplicado por 2.

```
while (numero <1000) {
print (numero)
numero <- numero * 2
}
```

Así pues, empezamos con un 1 y, en cada vuelta, vamos doblando este resultado. Cuando este resultado exceda el número que hemos fijado aquí, es decir el 1000, saldremos del while y seguiríamos con nuestro programa.

Ejecutamos y lo que vemos es que nos ha hecho 1, 2, 4, 8 ... Nos lo imprimiendo, porque le hemos pedido, y nos imprime hasta el 512, ya que tenemos un 256, que multiplicado por 2 es 512. El 512 cumple esta propiedad. Imprime el 512, lo multiplica por 2 y el guarda. Este valor aquí ahora vale 1024. Como a 1024 ya no cumple esta condición, no entra dentro de este espacio de aquí dentro y sale del bucle.

¡Continuamos!

A continuación proponemos algunos ejercicios sobre bucles. Encontrarás la solución de cómo hacerlo más adelante:

1. Explora el uso de la instrucción break. Programa un bucle que recorra los números del 1 al 100 mientras los imprime en pantalla. Mediante una condición, dile que ejecute break cuando el número en el que se encuentre sea el 30. ¿Qué ha pasado?
2. Crea un bucle que itere sobre un vector que hayas definido tú, que tenga tanto números positivos como negativos. Para los números positivos, queremos que diga si son pares o no. Para los negativos, que los imprima en pantalla en sentido positivo, si son superiores a -10. Si el número es menor que -10, queremos que el bucle pare.
3. Crea dos números, los que quieras, y, mediante un bucle while, ve doblando a ambos en cada iteración. Detener el bucle, cuando el mayor sea menos 1000 unidades mayor que el pequeño. Piensa detenidamente en cómo puedes definir esta condición de una manera sencilla, para que el bucle se ejecute mientras no se cumpla.

2.10 RESOLUCIÓN DE EJERCICIOS: BUCLES

¡Bienvenidos de nuevo!

Veamos cómo se resuelven los ejercicios de bucles.

El primero de todos, que es el más sencillo, se basa en crear una estructura de bucle for simple, en la que le pedimos que, cuando encuentre el valor y = 30, salga del bucle. ¿Qué pasará cuando ejecutamos esto? Que irá imprimiendo todos los números del 1 al 100 hasta que encuentre el 30. A partir de ahí, saldrá del bucle. Esto implica que imprimirá el 1 hasta el 29 y el 30 no llegará a imprimirlo, porque saldrá en este momento aquí.

```
for (i in 1: 100) {
if (y == 30) {
break
}
print (i)
}
```

Si lo ejecutamos, podremos ver efectivamente que llegamos hasta el número 29.

El segundo ejercicio es más rebuscado. Lo que buscamos es definir un vector de números positivos y negativos, y aplicarles una cierta acción o una cierta otra, en función de su valor. Lo primero que buscamos es si el número es positivo. En este caso, realizamos todas estas acciones de aquí dentro.

```
vectordefinido <- c (1, -1,3,4, -5.20, -12,4,5)
for (i in vectordefinido) {
```

```
if (i> = 0) {
if (y %% 2 == 0) {
print ( "Par")
} Else {
print ( "Impar")
}
} Else {
if (y <(- 10)) {
break
}
print (-y)
}
}
```

En función de si es par, utilizando esta sintaxis aquí (y %% 2 == 0), o si no lo es, es decir, de lo contrario. Aquí lo único que hacemos es imprimir, si es par, si cumple la primera condición, estando dentro de los números positivos; o, de lo contrario, imprimimos que es impar.

Si no estamos en el caso de números positivos, entramos esta parte de aquí al final.

```
else {
if (y <(- 10)) {
break
}
print (-y)
}
```

Aquí, lo que hacemos es comprobar si el número es menor que -10. Si el número es menor que -10, es decir, -11, -12, etc., salimos del bucle. En caso contrario, mostramos el número cambiado de signo, que se puede hacer sencillamente añadiendo un menos ("-") frente al valor sobre el que ITER. Ejecutamos el vector, el bucle, y vemos que se ejecuta correctamente, ya que obtenemos "Impar", el número en positivo, "Impar", "Par", el número en positivo, "Par"; y este número de aquí y los subsecuentes (-12,4,5) ya no aparecen, ya que hemos accedido aquí (break) y hemos roto el bucle.

Y, ya para terminar, lo que buscamos es crear dos números

```
a <- 2
b <- 1
```

y, utilizando un bucle while, buscamos si la diferencia entre estos números (yo lo he hecho utilizando la función "absoluto" (abs), que lo que hace es que nos calcula la diferencia sin importar si el mayor es el primero o el segundo), si esta diferencia es menor que 1000.

```
while (abs (a-b) <1000)
```

Mientras esto se cumpla, es decir, mientras la diferencia entre a y b sea menor que 1000, vamos doblando los números utilizando estas instrucciones aquí.

```
a <- a * 2
b <- b * 2
```

Una vez se cumpla, salimos del bucle y imprimimos los resultados.

```
print (a)
print (b)
```

lo ejecutamos

```
while (abs (a-b) <1000) {
a <- a * 2
b <- b * 2
}
print (a)
print (b)
```

y vemos que, al ser potencias de 2, lo que estamos obteniendo es 2048 y 1024.

Podemos hacer las pruebas con otros valores y veríamos cuando se cumple que esta diferencia, es decir, que el resto entre a y b sea mayor que 1000.

La ventaja de utilizar esto aquí (abs (a-b)) es que, si yo pongo un 1 y pongo un 5, ejecuto estas variables, obtengo los mismos resultados que no hubiera obtenido si hubiera hecho a - b directamente.

Y eso es todo, ¡seguimos!

2.11 IDEAS CLAVE: PROGRAMACIÓN CON R

Ya estamos en el final del segundo módulo. A continuación veremos las ideas y los conceptos más importantes que hemos desarrollado:

- Los diferentes tipos de objeto que se pueden crear en R son:
  - Variables unitarias: utilizadas para guardar valores únicos del tipo que sea.
  - Vectores: conjuntos de variables unitarias del mismo tipo.
  - *Dataframes*: estructuras similares a las hojas de cálculo de Excel.
  - Listas: parecidos a los vectores, pero más flexibles.

- Hemos aprendido cómo funcionan las estructuras condicionales y las comparaciones en R.
    - Por un lado, podemos utilizar *if* y *else*, los condicionales más clásicos para definir procesos diferenciados.
    - Por otro lado, la función compacta *ifelse*, que permite condensar este mismo proceso binario en una única función.

- Hemos visto cómo utilizar las funciones ya existentes en R y qué estructura general siguen, así como una breve introducción a la creación de funciones propias, lo que nos permitirá reproducir procesos repetitivos de análisis muy fácilmente.

- Por último, hemos aprendido a diseñar y aprovechar estructuras en bucle, es decir, procesos iterativos (como el *for* o el *while*) sobre nuestros objetos (listas, dataframes o vectores) y cómo podemos especificar sus condiciones de parada.