PROGRAMACIÓN Y MANEJO DE BASE DE DATOS EN R - CON UN ENFOQUE A LA ECONOMÍA

César Anderson Huamaní Ninahuanca

2021-07-12

Contents

Prólogo

Este libro es una guía para los estudiantes y egresados de economía, así también como para las personas que desean aprender a manejar el software R con aplicaciones a la economía.

Si tienes como objetivo desarrollarte profesionalmente en el análisis de datos, aprender R es fundamental. En el mercado actual las empresas requieren personas con conocimiento en softwares de programación que permitan tratar grandes cantidades de datos de manera versátil y eficientemente, es decir, programar funciones y tareas que disminuyan el uso de tiempo y recursos en su tratamiento. Al ser R un lenguaje orientado en objetos, cualquier problema que imagines se podrá sistematizar a través de la creación de algoritmos o funciones.

R tiene enormes ventajas en relación a otras softwares usados por estudiantes y egresados de economía como: STATA, EVIEWS, SPSS y otros. R es muy versátil que puede desarrollar todos los procedimientos de los otros softwares, incluso más. En R puedes crear dashboards, páginas web, informes automatizados, scrapear páginas web, entre otras muchas más tareas.

Asimismo, el libro nace como respuesta a las asimetrías de información entre los estudiantes y egresados de la capital y de provincias, el objetivo principal del libro es cerrar brechas educativas entre estudiantes o egresados de provincias con personas del mismo perfil que residen en la capital u otras regiones con estudios en universidades en donde se imparte cursos de programación en R.

El libro consta de 20 capítulos: El capítulo 1, **"R como calculadora"** en donde se muestran las principales operaciones matemáticas y estadísticas, así también como una introducción a la creación de funciones lo que permitirá ver el enorme potencial del software; en el capítulo 2, **"Vectores"** se da a conocer las clases de vectores que existen en R, así también como las operaciones, lógicas, matemáticas y estadísticas entre vectores, por último se muestra como seleccionar elementos y subvectores; en el capítulo 3, **"Listas y Factores"** se muestra como crear listas y factores y su uso en temas posteriores; en el capítulo 4 **"Matrices"** se desarrolla como crear matrices y sus operaciones lógicas, matemáticas y estadísticas, así también como la selección de elementos y operaciones de cálculo matricial.

1 R COMO CALCULADORA.

Uno de los primeros pasos para poder dominar **R** es comprender su uso como si fuera una calculadora. **R** realiza una variedad de operaciones matemáticas y lógicas de una manera muy sencilla. En las siguientes líneas se podrá observar algunas operaciones matemáticas[1] básicas.

1.1 Principales operaciones matemáticas.

Desarrollar calculos matemáticos en **R** es muy similar que usar una calculadora, a continuación mostramos los principales operadores aritméticos y su sintaxis en el entorno **R**.

OPERACIÓN	SINTAXIS
Adición	+
Sustracción	-
Producto	*
División	/
División: Para Calcular el cociente	%/%
División: Para Calcular el residuo	%%
Potencia	^ **

Veamos unos ejemplos de como usarlo.

Si deseamos sumar 4 más 3 en R. Tenemos que digitar en la consola `4+3`.

```
4+3    # Adición.
```

```
## [1] 7
```

Como se puede dar cuenta el resultado es 7. Si deseamos realizar la sustracción de 5 en 7. Tenemos que digitar en la consola la operación `5-7`.

```
5-7    # Sustracción.
```

```
## [1] -2
```

Para el caso de la multiplicación se usa el operador `*`, es así que si deseamos multiplicar 3 por 4, entonces, tendremos que digitar en la consola `3*4`.

```
3*4    # Producto.
```

```
## [1] 12
```

Si se desea calcular la división usamos el operador `/`.

```
6/2    # División.
```

```
## [1] 3
```

En efecto, al dividir 6 entre 2 nos resulta 3. Pero si se desea calcular sólo el cociente de una división inexacta usaremos el operador `%/%`.

```
10%/%3   # División para encontrar el cociente.
```

```
## [1] 3
```

El cociente resultante es 3. Asimismo, si sólo desea obtener el residuo de una división inexacta tendrá que usar el operador `%%`.

[1] Si usted no comprende esta sintaxis, le recomendamos revisar el capítulo de vectores aleatorios o buscar cuales son los argumentos de la función `sample()`.

```r
10%%3    # División para encontrar el residuo.
```

```
## [1] 1
```

En efecto, el residuo de dividir 10 entre 3 es 1.

Otra operación fundamental es elevar un número a la potencia n. Veamos un ejemplo en elevar 4 al cubo.

```r
4^3    # Potencia.
```

```
## [1] 64
```

Usted se puede dar cuenta que hemos usado el operador `^`, pero este no es el único operador que calcula la potencia de un número, también se puede calcular si se usa el operador `**`.

```r
4**3   # Potencia.
```

```
## [1] 64
```

Entonces, puede usar `^` o `**` si desea calcular la potencia de un número.

Ahora veremos algunas operaciones matemáticas un poco más avanzadas, que nos servirán en un futuro para poder hacer cálculo estadístico, matemático y poder transformar variables cuando especifiquemos modelos econométricos.

OPERACIÓN	SINTAXIS
Factorial	**factorial()**
Combinatorio	**choose()**
Raíz Cuadrada	**sqrt()**
Número de Euler-Exponente	**exp()**
Logaritmo Natural	**log()**
Valor Absoluto	**abs()**

Si deseamos calcular el factorial de un número usaremos la función `factorial()`. Veamos un ejemplo del factorial de 5.

```r
factorial(5)     # Factorial.
```

```
## [1] 120
```

Nos resulta 120, recordar que el factorial de un número es la multplicación de $n(n-1)(n-2)...1$. En el caso del factorial del 5, se multiplicó $5*4*3*2*1$.

Asimismo, si deseamos calcular las combinaciones posibles de un número agrupado por otro, usaremos el combinatorio. En R se usa la función `choose()`.

```r
choose(5,3)    # Combinatorio.
```

```
## [1] 10
```

R nos arroja el valor de 10, recordar que el cálculo que se usó fue el siguiente: $\frac{5!}{2!*3!}$.

Si tenemos la varianza de una variable, podremos calcular su desviación si le sacamos la raíz cuadrada. Entonces, si deseamos calcular la raíz cuadrada usaremos la función `sqrt()`.

```r
sqrt(12)     # Raíz cuadrada.
```

```
## [1] 3.464102
```

Nos resulta 3.464102, es un número con 6 decimales y es posible que usted lo desee con más o menos decimales. En la siguiente sección veremos como mostrar los decimales que deseados.

Otra posible situación a la que se podrá enfrentar en un futuro es si desea calcular los Odds ratios de los modelos logit, recordemos que estos se calculan si se les saca el exponencial. Entonces si usted desea calcular el exponencial de un número tendrá que usar la función `exp()`.

```
exp(1)      # Si se considera 1, dará el número de euler.
```

```
## [1] 2.718282
```

En este ejemplo se ha calculado el exponencial de 1, lo que nos da el número de Euler que es 2.7182818. Asimismo, si deseamos especificar nuestro modelo econométrico en logaritmos naturales y así poder hacer una interpretación a nivel de elasticidades tendremos que transformar las variables aplicándoles el logaritmo natural, para lo cual tendremos que usar la función `log()`.

```
log(3)      # Logaritmo Natural.
```

```
## [1] 1.098612
```

Por otro lado, si deseamos calcular el valor absoluto de un número usaremos la función `abs()`.

```
abs(-2)    # Valor Absoluto.
```

```
## [1] 2
```

Ahora veremos como calcular las razones trigonométricas, para lo cual en la tabla siguiente se muestra la sintaxis de las razones trigonométricas.

RAZÓN TRIGONOMÉTRICA	SINTAXIS
seno	**sin()**
coseno	**cos()**
tangente	**tan()**

Para poder calcular las razones trigonométricas se tiene que considerar al **ángulo en pi radianes**. En **R** el valor de pi se puede obtener si digitamos en la consola `pi`, R comprende que al escribir `pi` se está llamando al número irracional `pi`.

```
pi
```

```
## [1] 3.141593
```

En efecto, nos resulta 3.1416.

Asimismo, como se puede dar cuenta para la cosecante, secante y cotangente no se tiene un función específica, ya que estan son las inversas del seno, coseno y tangente, respectivamente.

Si deseamos calcular el seno de 30. Entonces tenemos que usar la función `sin()` y especificar el ángulo en **pi radianes**.

```
sin(pi/6)   # Seno de 30 grados sexagesimales.
```

```
## [1] 0.5
```

A continuación veamos unos ejemplos para el coseno y la tangente.

```
cos(pi/4)   # Coseno de 45 grados sexagesimales.
```

```
## [1] 0.7071068
```

```
tan(pi/4)   # Tangente de 45 grados sexagesimales.
```

```
## [1] 1
```

Ahora si se desea trabajar omitiendo los pi radianes, se tendrá que usar las funciones **sinpi()**, **cospi()**, **tanpi()**.

A continuación se muestran algunos ejemplos:

```
sinpi(1/6)   # Seno de 30 grados sexagesimales.
```

```
## [1] 0.5
```

En efecto, es el mismo resultado que se obtuvo con la función `sin()`, solo que aquí se está omitiendo los pi radianes.

```
cospi(1/4)   # Coseno de 45 grados sexagesimales.
```

```
## [1] 0.7071068
```

```
tanpi(1/4)   # Tangente de 45 grados sexagesimales.
```

```
## [1] 1
```

1.2 Decimales exactos y redondeo de cifras.

1.2.1 Número de cifras.

Si al realizar los cálculos usted desea obtener los resultados con un número determinado de cifras se tendrá que usar la función `print()`[2] en donde se tendrá que especificar el número de dígitos que se desea.

Por ejemplo deseamos obtener la raíz cuadrada de 12 pero queremos que el resultado se muestre 10 cifras. Entonces, usaríamos la siguiente sintaxis.

```
# La raíz cuadrada de 12, pero que nos muestre 10 cifras,
print(sqrt(12), 10)
```

```
## [1] 3.464101615
```

En efecto, nos muestra 10 cifras. A continuación se muestra un ejemplo adicional en donde le indicamos a **R** que nos muestre el mismo resultado pero ahora sólo con 3 cifras.

```
# La raíz cuadrada de 12, pero que nos muestre 3 cifras,
print(sqrt(12), 10)
```

```
## [1] 3.464101615
```

En efecto, tenemos la raíz cuadrada de 12 en 3 cifras.

Es importante saber que el número máximo de cifras que reporta **R** usando la función `print()` es de 17. Veamos un ejemplo, sabemos que pi es un número irracional por lo que tiene infinitos decimales. Entonces, si queremos mostrar 16 decimales, usaríamos la siguiente sintaxis.

```
print(pi, 17)
```

```
## [1] 3.1415926535897931
```

En efecto, nos arroja 16 decimales. ¿Por qué usamos 17 y no 16 en la función? ya que `pi` tiene un entero y queremos 16 decimales, se tiene que especificar 17. Pero si por ahí se le ocurriera mostrar `pi` con 20 decimales, usted usaría la siguiente sintaxis.

```
print(pi, 21)
```

```
## [1] 3.1415926535897931
```

[2] El ciclo while se desarrollará en el siguiente capítulo.

Pero lamentablemente, sólo obtiene el mismo número de decimales que si hubiera usado `print(pi, 17)`. Con lo cual queda demostrado que la función `print()` sólo puede mostrar hasta 17 cifras.

Pero no se frustre, en R hay otras funciones que permiten obtener los resultados con más decimales. Una de estas funciones es `sprintf()`, el cual puede mostrar más funciones pero los decimales después del 15 no serán tan exactos[3].

Veamos un ejemplo, en donde deseamos que se muestren 50 decimales

```
sprintf("%.50f",pi)
```

```
## [1] "3.14159265358979311599796346854418516159057617187500"
```

Por otra parte, si quieres trabajar con muchos decimales puede resultar tedioso usar en cada cálculo la función `print()` o `sprintf()`. Para solucionar esto se usa la función `options()`, por ejemplo si deseas que los calculos que vas a desarrollar se trabajen con 10 decimales tienes que correr en la consola la siguiente sintaxis.

```
options(digits=10)
```

Y con eso todos los resultados que se calculen se mostraran con 10 cifras. Asimismo, es preciso aclarar que esta función puede arrojar como máximo de 22 dígitos.

1.2.2 Redondeo.

Si se desea redondear una operación se tendrá que usar la función `round()`, que al igual que el anterior se tendrá que especificar el número de dígitos, que en este caso será el número de decimales a los que se desea redondear.

A continuación se muestra un ejemplo:

```
# La raíz cuadrada de 3 redondeada a cinco decimales.
round(sqrt(3), 5)
```

```
## [1] 1.73205
```

Sí sólo se considera la función `round()` redondeará a la cifra entera:

```
# La raíz cuadrada de 3 redondeada a la cifra entera.
round(sqrt(3))
```

```
## [1] 2
```

1.3 Guardar o definir objetos.

Hasta ahora sólo hemos hecho calculos y sólo los hemos mostrado en la consola. Pero puede surgir la necesidad de guardar estos cálculos para que en un futuro podamos generar cálculos más complejos o llamar a estos resultados. Entonces, surge la necesidad de guardar nuestros resultados.

En **R** todo es un objeto, desde un número, un vector, una variable, una matriz, un data frame, una lista, un factor, un gráfico, etc. Así que lo primordial será guardar estos objetos. Para lo cual tendremos la opción de usar uno de estos tres operadores si deseamos guardar estos objetos en la memoría del software.

Estos operadores son: `<-`, `=`, `->`.

Por ejemplo si deseamos guardar un objeto que se llame `a` y que tome el valor de `8`. Entonces, nosotros definiremos al objeto `a` de la siguiente manera.

[3]Para más detalles puede revisar la siguiente documentación: https://cutt.ly/hjfCXw5

```
a<-8
```

Hemos usado uno de los operadores (`<-`) para definir objetos[4]. Ahora si deseamos llamar a este objeto, nosotros sólo digitamos el objeto en la consola y damos enter.

```
a
```

```
## [1] 8
```

En efecto, podemos ver que el objeto `a` toma el valor de 8. Podemos usar los otros operadores indiferentemente para poder definir objetos. Veamos un pequeño ejemplo.

```
b<-6
d=6
6->f
```

En este caso hemos definido a los objetos `b`, `d` y `f` con el valor de 6 a cada uno.

Veamos si los valores son los correctos.

```
b
```

```
## [1] 6
```

```
d
```

```
## [1] 6
```

```
f
```

```
## [1] 6
```

En efecto, cada objeto toma el valor de 6.

Como se ha podido dar cuenta se abre una amplia gama de nombres con los que usted podría guardar objetos. Por ejemplo, podemos usar la palabra `objeto` para guardar la suma de `5+12`.

```
objeto<-5+12
```

Llamando a nuestro objeto `objeto`.

```
objeto
```

```
## [1] 17
```

Como podemos ver, ahora `objeto` toma el valor de 17.

Pero como en todo software, los objetos no pueden tomar cualquier nombre. Es así que hay una regla única, que se debe tener en cuenta a la hora de guardar objetos, estos **"NO pueden empezar con números"**.

Veamos un ejemplo en donde intentamos definir un objeto en donde el nombre de este empiece con un número.

```
1y<-4
```

```
## Error: <text>:1:2: unexpected symbol
## 1: 1y
##      ^
```

Vemos que nos sale un mensaje de error, que nos dice que hay un símbolo inesperado en `1y` el cual es el `1`. Entonces, para definir objetos, estos nunca tienen que empezar con un número.

[4]El operador más usado y adecuado para definir objetos es `<-`, pero el uso de los otros operadores es más usado en casos especiales, estos casos lo veremos más adelante.

1.4 Funciones.

Entonces, una vez que ya sabemos definir objetos, veremos otro de los objetos fundamentales en R, estos son las funciones[5]. Una función realiza un proceso o algoritmo que ha sido programado con anterioriedad.

Los creadores de R definieron funciones bases cuando crearon el software. Por ejemplo, una función base es `sqrt()`, el cual calcula la raíz cuadrada de un número. Nosotros también podemos definir nuestras propias funciones, al igual que lo han programado los creados de **R**. Esto es lo fundamental y maravilloso de los softwares de programación, ya que nos permite personalizar y adaptar a nuestras necesidades el software.

Veamos a manera de introducción cómo definir una función en R. Para esto se tendrá que hacer uso de la función `function()` en donde se tendrá que especificar argumentos que definiran la función que desemos crear.

Vamos empezar definiendo funciones como si fueran expresiones matemáticas. Por ejemplo, deseamos definir el objeto `y` que es función del argumento o variable `x`, es decir, `x` será el dominio y `y` será el rango.

```
# Se define la función "y" que tiene como argumento a  "x"
# que tiene una forma cuadrática: x^2 +2

y<-function(x){
  x^2+2
}
```

Hemos creado el objeto `y` que es una función que tiene como argumento o variable a `x`. Es decir, cada vez que cambiemos el valor de `x` cambiará el valor de `y`, ya que `x` define a `y`.

Veamos como podemos usar esta función.

```
# Para poder evaluar la función "y" en un determinado valor de "x"
# En este caso x=0, la sintaxis sería la siguiente:

y(x=0)
```

```
## [1] 2
```

Como se puede observar si la función se evalua en cero, es decir, si `x` es igual a 0; la función tomará el valor de 2 ($0^2 + 2 = 2$).

Veamos otro ejemplo para que quede más claro. Ahora veremos que valor toma la función si `x` es igual a `8`.

```
y(x=8)
```

```
## [1] 66
```

Nos resulta el valor de 66 ($8^2 + 2 = 66$).

Si se tiene en consideración una función en R^3. Se tendá que tomar en cuenta dos argumentos o variables dentro de la función `function()`. como se muestra a continuación:

```
# Se define la función Z que depende de la variable x e y.
z<-function(x,y){
  x+4+4*y
}
```

Se ha definido a la función `z` como una función de `x` y `y`. Usted puede estar pensando que el argumento `y` es la función que hemos definido anteriormente. Pero, déjeme decirle que no es así. Cuando se define funciones en R los argumentos de estás no toman las propiedades de los objetos guardados en la memoría del software. Es así que el nombre de los argumentos puede ser incluso el nombre de objetos que están guardados en la

[5]En esta parte del libro, veremos sólo una parte introductoria de funciones, en el capítulo 9 veremos a más detalle y profundidad este tema.

memoría, pero esto no implicará que el argumento juege el rol del objeto que está definido en la memoría del software. En palabras sencillas, el argumento `y` en la función `z` no tiene nada que ver con la función `y` que definimos anteriormente.

Veamos que resulta en la función `z` si `x` toma el valor de 0 y `y` el valor de 2.

```
# Se evalua la función "z" cuando "x" e "y" toman el valor
# de 0 y 2, respectivamente.
z(x=0,y=2)
```

```
## [1] 12
```

Podemos ver que el resultado es 12 $(0 + 4 + 4 * 2 = 12)$.

Para que quede claro, a continuación se muestra un ejemplo pero considerando la densidad de la función de distribución normal.

```
# Se define la función "N" que es la densidad de la función
# de distribución normal con media igual a 0.5 y
# desviación estándar de 0.1.

N<- function(x){
  dnorm(x, mean =0.5, sd=0.1)
}

# Se evaluará cuando la variable aleatoria toma el valor de 0.2.

N(0.2)
```

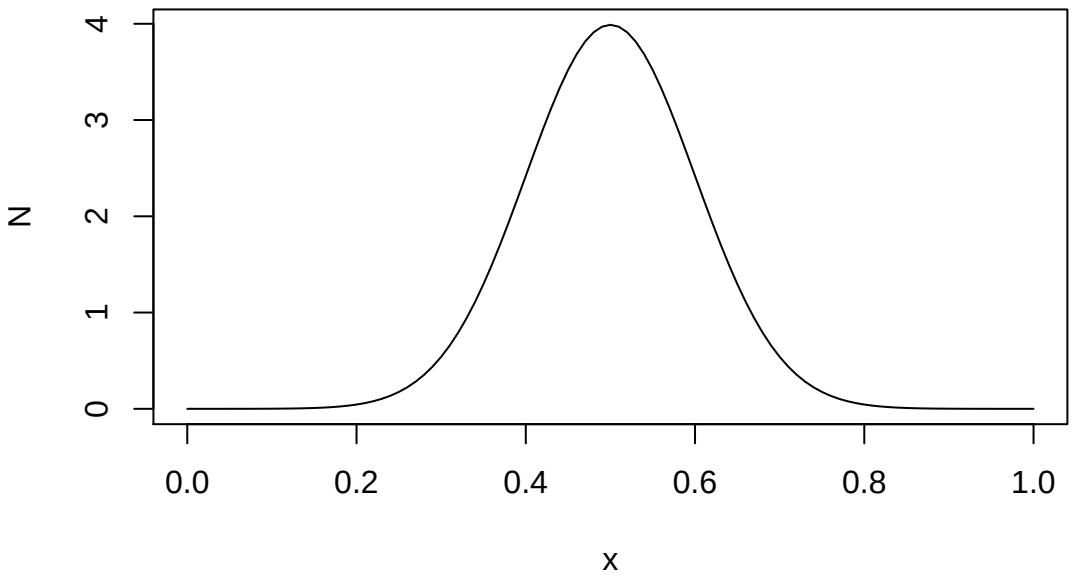
```
## [1] 0.04431848
```

Es preciso aclarar que para poder obtener la densidad de la distribución normal se uso la función `dnorm()`, para mejor detalle podría revisar la documentación si digita en la consola `?dnorm` y da enter a continuación.

Hemos visto una pequeña introducción sobre funciones, hasta el momento. En el capítulo 9, veremos a más detalle cómo programar funciones, pero para esto es necesario aprender más cosas que nos permitan programar cualquier tipo de funciones. Es así que en los capítulos posteriores aprenderemos más objetos como: vectores, matrices, listas, etc.

Por el momento como motivación y ver lo que se viene en los siguientes capítulos. Haremos un gráfico de la función de densidad `N` que hemos definido anteriormente. Y podrá ver con que facilidad se logra este objetivo.

```
# La gráfica de la densidad de la FDN.
plot(N)
```



2 VECTORES.

Uno de los objetos más importantes del entorno **R** son los vectores. Un vector en **R** es un objeto que agrupa a varios elementos, es algo muy similar a un conjunto. Gracias a ellos se pueden construir matrices, listas, data frame, etc. Lo que posibilita que se pueda hacer análisis de datos al más alto nivel. Por tanto, en las siguientes lineas se muestra como definir vectores, sus clases, y las funciones más importantes para poder obtener vectores y realizar operaciones matemáticas, lógicas y estadísticas.

2.1 Definición de un vector.

Para poder definir un vector se puede usar cualquiera de los tres operadores `<-`, `=`, `->`. En este libro, de aquí para adelante, para definir vectores usaremos el símbolo `<-`.

Tambien es necesario la función `c` que sirve para concatenar elementos (más adelante se verá a más detalle esta función). Con estas dos herramientas es suficiente para poder definir vectores.

2.2 Clases de vectores.

En R, al igual que en otros softwares los objetos son de distintas clases. Esto permite un orden a la hora de transformar o generar operaciones, ya que todo vector, no puede aceptar operaciones matemáticas, estadísticas o lógicas.

En la siguiente tabla se muestran las clases de vectores que se pueden usar en **R**.

CLASE DE VECTOR	QUE CONTIENE
Integer	Números enteros

CLASE DE VECTOR	QUE CONTIENE
Numeric	Números reales
Logical	Caracteres lógicos
Character	Caracteres o Palabras
Complex	Números complejos

Como se mencionó anteriormente, para definir vectores usaremos el operador de definición `<-` y la función `c()`. esta función permite juntar elementos en un determinado objeto.

Para la clase de vectores **Integer** sólo permite elementos que sean números enteros. En el entorno **R** para definir un número entero es necesario agregar al número la letra **L**, como se muestra a continuación:

```
#Vector Integer - Números enteros.
x<-c(1L,2L,3L,4L,5L,6L)
y<-c(1L,3L,5L,7L,9L,11L,13L)
```

Se realizá la definición de los vectores **x** y **y**. En los cuales se hizo uso de `<-` y `c()`. Los resultados se muestran a continuación:

```
x
```

```
## [1] 1 2 3 4 5 6
```

```
y
```

```
## [1]  1  3  5  7  9 11 13
```

En efecto, podemos ver que `x` vale 1, 2, 3, 4, 5, 6 y `y` vale 1, 3, 5, 7, 9, 11, 13.

Con respecto a la clase de vectores **Numeric** estos adminten números reales. Es muy fácil definirlos, a continuación se muestra un ejemplo.

```
#Vector Numeric - Números reales.
z<-c(1.3, pi, exp(1))
t<-c(sin(pi/4), log(45), tan(pi/3))
```

Podemos ver que los vectores **z** y **t** contienen números reales, en el caso del vector **z** está compuesto por 1.3, pi y el número de euler. Los resultados son los siguientes:

```
z
```

```
## [1] 1.300000 3.141593 2.718282
```

```
t
```

```
## [1] 0.7071068 3.8066625 1.7320508
```

Un caso especial y fundamental de los lenguajes de programación son los vectores lógicos, estos permiten que los procesos sean más rápidos y a la vez más intuitivos. Para el caso de estos vectores, sólo acepta como elementos a `TRUE` y `FALSE`. A continuación se muestra un ejemplo:

```
# Vector Logical - Caracteres lógicos.
m<-c(TRUE,FALSE,FALSE,TRUE)
p<-c(T,F,F,T,T,T,F)
```

Como se habrán podido dar cuenta no es necesario escribir las palabras completas (`TRUE` o `FALSE`) es suficiente con escribir sus iniciales. Para comprobarlo a continuación se muestran los resultados.

```
m
```

```
## [1]  TRUE FALSE FALSE  TRUE
```

```
p
```

```
## [1]  TRUE FALSE FALSE  TRUE  TRUE  TRUE FALSE
```

Si se desea crear vectores **Character** se tendrá que usar como elementos sólo palabras o caracteres. Estos vendrán especificados por las comillas "", un ejemplo se muestra a continuación:

```
# Vector Character - Palabras.
p1<-c("Luis", "María", "José")
p2<-c("12", "casa", "pi")
```

El vector p1 contiene nombres (palabras) y el vector p2 contiene los números 12 y pi, pero como se usó las comillas, **R** los considera como caracteres, adicionalmente tiene a la palabra casa.

```
p1
```

```
## [1] "Luis"  "María" "José"
```

```
p2
```

```
## [1] "12"   "casa" "pi"
```

Por último, los vectores **complex** sólo aceptan elementos que sean números complejos. Para definir un número complejo es necesario considerar el número imaginario **i**. A continuación se muestran uno ejemplo:

```
# Vector Complex - N?meros complejos.
c1<-c(1+2i, 4i, 3+6i)
```

El vector c1 contiene tres elementos, los cuales son números complejos. Los resultados son los siguientes:

```
c1
```

```
## [1] 1+2i 0+4i 3+6i
```

2.2.1 ¿Cómo saber si un objeto es un vector?

Para poder saber si un objeto es un vector, es necesario usar la función `is.vector()`. Este nos arrojará los valores TRUE (si es un vector) o FALSE (si no es un vector). A continuación se muestran los resultados para algunos de los vectores que hemos definido antes:

```
is.vector(x)
```

```
## [1] TRUE
```

```
is.vector(z)
```

```
## [1] TRUE
```

```
is.vector(m)
```

```
## [1] TRUE
```

```
is.vector(p1)
```

```
## [1] TRUE
```

```
is.vector(c1)
```

```
## [1] TRUE
```

Como era de esperarse, todas las variables definidas, resultan ser vectores.

2.2.2 ¿Cómo saber de que clase es un vector?

Para conocer la clase, se tendrá que usar la función `is.integer()`, `is.numeric()`, `is.logical()`, `is.character()` y `is.complex()`. Al igual que antes este arrojará `TRUE` o `FALSE`. A continuación se muestran unos ejemplos.

```
is.integer(x)
```

```
## [1] TRUE
```

```
is.numeric(x)
```

```
## [1] TRUE
```

```
is.logical(m)
```

```
## [1] TRUE
```

```
is.character(p1)
```

```
## [1] TRUE
```

```
is.complex(p1)
```

```
## [1] FALSE
```

Puede parecer algo confuso, pero usted se puede preguntar, por qué el vector **x** que fue definido como **Integer** también arroja como si fuera un vector **Numeric**. Esto es porque los números enteros están contenidos en los números reales, mejor dicho el conjunto de números enteros es un subconjuto del conjunto de los números reales.

2.2.3 ¿Qué pasa si defino un vector con distintas clases de elementos?

¿Qué clase de vector será si defino el siguiente vector?

```
v<- c(12, "azul", 2+1i, pi)
is.integer(v); is.numeric(v); is.logical(v); is.character(v); is.complex(v)
```

```
## [1] FALSE
```

```
## [1] FALSE
```

```
## [1] FALSE
```

```
## [1] TRUE
```

```
## [1] FALSE
```

Como se puede observar el vector fue definido como **Character**, porque existe una jerarquía de elementos. Si el vector contiene un elemento caracter considerará a los otros elementos como caracteres. Como se muestra a continuación:

```
v
```

```
## [1] "12"               "azul"             "2+1i"             "3.14159265358979"
```

La jerarquía es la siguiente:

`Character>Complex>Numeric>Integer>Logical`

Entonces, si quitamos el elemento caracter el vector debe de convertirse en un vector complejo. Veamos:

```
v<- c(12, 2+1i, pi)
is.integer(v); is.numeric(v); is.logical(v); is.character(v); is.complex(v)
```

```
## [1] FALSE
```

```
## [1] FALSE
```

```
## [1] FALSE
```

```
## [1] FALSE
```

```
## [1] TRUE
```

En efecto, el vector es un vector **complejo**.

2.3 Número de elementos de un vector.

Si deseamos conocer el número de elementos o dimensión o tamaño que tiene un vector. Nosotros usaremos la función `length()`. La cual nos permitirá conocer el número de elementos por los que está compuesto el vector. Veamos el siguiente ejemplo.

```
# Definimos el vector.
wasu<-c(1,3,54,6,3,4,2,4,6,9,87,8,4,5,3,2,2,3,4,5,6,4,5,6,7,7,7)
wasu
```

```
##  [1]  1  3 54  6  3  4  2  4  6  9 87  8  4  5  3  2  2  3  4  5  6  4  5  6  7  7  7
```

Y si deseamos saber cuántos elementos tiene el vector `wasu` entonces:

```
length(wasu)
```

```
## [1] 27
```

El vector `wasu` tiene 27 elementos.

Veamos un ejemplo adicional.

```
length(c("Marcos","Rocio",12,12,12,3434,5656,788,5,4, TRUE,F))
```

```
## [1] 12
```

El vector que acabamos de crear tiene 12 elementos.

2.4 Funciones para construir vectores más eficientemente.

2.4.1 Vectores de elementos consecutivos:

Para poder construir un vector de números consecutivos se necesitará el operador `:`. Por ejemplo, si deseamos contruir la serie de números desde el 1 al 10, entonces la sintaxis sería la siguiente.

```
# Vector que tiene como elementos desde el 1 al 10.
x0<-1:10
x0
```

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

Otro ejemplo sencillo, podría ser si deseamos generar números consecutivos desde el 20 al 45.

```
# Vector que tiene elementos desde el 20 al 45.
x1<-20:45
x1
```

```
##  [1] 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45
```

En efecto, se ha generado la serie de números consecutivos desde el 20 al 45.

2.4.2 Vectores con patrones:

También existen vectores en donde los elementos siguen cierto patrón, como una secuencia aritmética o patrónes recurrentes. Veremos 2 funciones importntísimas para crear vectores los cuales son `rep()` y `seq()`.

FUNCIÓN	GENERA
rep()	Repite elementos o vectores
seq()	Elementos ordenados en progresión aritmética

Veamos algunos ejemplos de uso.

2.4.2.1 Función rep().

Est función tiene la siguiente sintaxis:

```
# rep(x, ...)
```

Donde:

`x`: Es el elemento o vector que se desea repetir.

`...`[6]: Son otros argumentos como: `times`, `each` y `length.out`.

Entonces, si deseamos construir un vector en donde se repita el `2` diez veces, podemos usar la siguiente sintaxis.

```
# Para repetir el 2 diez veces.
rep(x=2,times=10)
```

```
##  [1] 2 2 2 2 2 2 2 2 2 2
```

Con la sintaxis anterior le indicamos a **R** que el elemento `x=2` se repita `times=10` veces.

Veamos otro ejemplo en donde el elemento `Luis` se repita `8` veces.

```
# Para repetir "Luis" ocho veces.
rep("Luis",8)
```

```
## [1] "Luis" "Luis" "Luis" "Luis" "Luis" "Luis" "Luis" "Luis"
```

Usted habrá notado que en esta ocación he omitido colocar el nombre de los argumentos `x` y `times` dentro de la función. Esto se hace siempre y cuando se sepa el orden de los argumentos. Es así que sí sé que el primer argumento de la función `rep()` es el elemento o vector que se desea repetir, entonces podría omitir de la sintaxis la `x`. Esto es así porque el orden de los argumentos de la función `rep()` es el siguiente:

```
rep(x, times = 1, length.out = NA, each = 1)
```

Como se ve, el primer argumento es `x`, el segundo es `times`, el tercero es `length.out` y el cuarto es `each`.

Veamos una situación en donde se coloca en orden inverso los argumentos. Por ejemplo queremos repetir el `8` cuatro veces.

```
# Estamos inviertiendo el orden intencionalmente.
rep(4, 8)
```

```
## [1] 4 4 4 4 4 4 4 4
```

Resultó que se repitió el elemento `4` ocho veces. Y no resulta lo que teníamos pensado el `8` cuatro veces.

Entonces, podemos usar el orden inverso siempre y cuando coloquemos los nombres de los argumentos. Por ejemplo:

[6] Si usted no comprende esta sintaxis, le recomendamos revisar el capítulo de vectores aleatorios o buscar cuales son los argumentos de la función `sample()`.

```r
rep(times=4, x=8)
```

```
## [1] 8 8 8 8
```

En efecto, ahora si conseguimos lo que hemos estado buscando, que el `8` se repita cuatro veces.

Entonces, como conclusión, si deseamos omitir el nombre de los argumentos debemos de asignar valores en el orden que fueron definidos.

Asimismo, otro detalle importante de las funciones es que los argumentos toman valores por defecto, por ejemplo el argumento `times` toma el valor de `1` que se representa `times=1`. Esto implica que si usted omite usar el argumento `times` en su sintaxis, `times` tomará el valor de `1`. Es decir, el elemento se repetirá 1 vez. Veamos el ejemplo.

```r
# Omitimos especificar el argumento times.
rep(8)
```

```
## [1] 8
```

En efecto, el elemento `8` se ha repetido una vez, es decir, se mantiene el 8.

Es así que es muy importante conocer los valores que toman por defecto los argumentos de las funciones. Con el uso y el mayor expertiz que vaya ganando en el uso del software R, usted podrá omitir nombres de argumentos ya que sabrá la posición de estos.

Ahora veamos como podríamos hacer repetir vectores.

El argumento `times` se usa cuando se quiere repetir todo el vector un número determinado de veces y `each` cuando se requiere repetir los elementos del vector. A continuación se muestran unos ejemplos.

```r
# Repetirir el vector 1,2,3,4,5 tres veces.
rep(1:5, times=3)
```

```
##  [1] 1 2 3 4 5 1 2 3 4 5 1 2 3 4 5
```

```r
# Repetirir los elementos del vector 1,2,3,4,5 tres veces.
rep(1:5, each=3)
```

```
##  [1] 1 1 1 2 2 2 3 3 3 4 4 4 5 5 5
```

```r
# Repetir los elementos del vector 1,2,3,4,5 tres veces y luego este nuevo vector dos veces.
rep(x0, each=3, times=2)
```

```
##  [1]  1  1  1  2  2  2  3  3  3  4  4  4  5  5  5  6  6  6  7  7  7  8  8  8  9  9  9 10
## [29] 10 10  1  1  1  2  2  2  3  3  3  4  4  4  5  5  5  6  6  6  7  7  7  8  8  8  9  9
## [57]  9 10 10 10
```

Para repetir Arbitrariamente los elementos del vector se usa la función concatenar `c` y se indica la veces que se desea repetir cada elemento del vector. A continuación un ejemplo:

```r
# Repetición arbitraria de elementos del vector 1,2,3,4,5,6
rep(1:6, times=c(3,0,1,0,4,2))
```

```
##  [1] 1 1 1 3 5 5 5 5 6 6
```

Se repite el elemento 1 tres veces, el elemento 2 cero veces, el elemento 3 una vez, el elemento 4 cero veces, el elemento 5 cuatro veces y el elemento 6 dos veces.

La función `rep()` también se puede usar para repetir el vector y especificar una determinada dimension. Con el argumento `length.out` podemos indicar la dimensión o tamaño deseado del vector.

El siguiente es un ejemplo de este caso:

```r
# Repetir los elementos del vector 1,2,3 dos veces.
# Pero la dimensión del vector resultante que sea de 5.
rep(1:3, each=2, length.out=5)
```

```
## [1] 1 1 2 2 3
```

Si no se hubiese especificado el argumento `length.out` el vector tendría una dimensión de 6, pero como se especificó que sólo tenga una dimensión de 5. Sólo considero los elementos hasta llegar a una dimensión o tamaño de 5.

A continuación se muestra un ejemplo más, con el fin de aclarar dudas.

```r
rep(1:10, each=5, length.out=3)
```

```
## [1] 1 1 1
```

Si no se hubiera especificado el argumento `length.out` el vector tendría un tamaño de 50, pero gracias a la especificación de `length.out` este sólo cuenta con un tamaño de 3.

2.4.2.2 Función seq(). Esta función tiene como fin que los elementos del vector aumenten o decrezcan en proporción a una razón aritmética. La sintaxis es la siguiente:

```
seq(from = 1, to = 1, by = ((to - from)/(length.out - 1)), length.out = NULL, along.with
= NULL, ...)
```

Donde:
`from`: Corresponde al número con el que empezará la progresión.
`to`: Indica el último número de la progresión.
`by`: Indica la razón aritmética.
`length.out`: La dimensión que se desea que tenga el vector resultante.
`along.with`: Con este argumento se le indica que tome la dimensión de otro objeto.

Un detalle importante de la función `seq()` es que no se puede usar mmutuamente el argumento `by` y el `length.out`, es decir, no se puede usar los dos argumentos a la vez. Veremos ejemplos de aquello líneas mas abajo.

Veamos un ejemplo, en donde se desea tener un vector que parta del 1 y termine en 21 y que aumente de 2 en 2.

```r
seq(1, 21, by=2)
```

```
##  [1]  1  3  5  7  9 11 13 15 17 19 21
```

En efecto tenemos el vector de número impares desde el 1 al 21.

Ahora veamos un caso donde la progresión tenga una razón aritmética negativa. Para este caso el `from` tiene que ser más grande que el `to` ya que la progresión es decreciente.

```r
seq(40, 3, by=-4)
```

```
##  [1] 40 36 32 28 24 20 16 12  8  4
```

En efecto, tenemos los números del 40 al 4 que decrecen de 4 en 4. El último elemento no es el 3, ya que el 3 queda fuera de la progresión aritmética, es decir, el valor de 4 es el mínimo elemento que se puede obtener en esta serie si la razón decrece de 4 en 4.

Si deseamos tener una determinada dimensión y omitir la razón se tendrá que especificar el argumento `length.out`. Ya que con esto le indicamos a R que calcule la razón aritmética, ya que nosotros deseamos que la secuencia tenga un determinado número de elementos.

A continuación veamos un ejemplo.

```r
# El vector empezará de 1 y terminará en 50
# y tendrá una dimensión de 10.

seq(1,50, length.out=10)
```

```
##  [1]  1.000000  6.444444 11.888889 17.333333 22.777778 28.222222 33.666667 39.111111
##  [9] 44.555556 50.000000
```

En efecto, al no asignarle la razón aritmética (`by`). El software ha calculado un `by` ya que nosotros le indicamos un tamaño específico.

También podemos usar el tamaño de otros objetos para poder indicar en la función `seq()` la dimensión deseada. Para esto tenemos que usar el argumento `along.with`.

Veamos un ejemplo a continuación.

```r
# Definimos el vector a:
a<-c(1,4,67,8)    # El vector a tiene 4 elementos.

# Entonces si queremos tener un vector de 4 elementos.
seq(1,10, along.with = a)
```

```
## [1]  1  4  7 10
```

En efecto, tenemos un vector de 4 elementos que parte desde 1 y termina en 10.

Por último, veremos el caso en donde sólo especificamos el `from` pero no el `to` y ningún otro argumento adicional.

R nos dice en la documentación de la función `seq()` que si sólo usamos el argumento `from` entonces devolverá una secuencia de números consecutivos desde el 1 hasta el número especificado en el from, es decir, `1:from`. El software incluso recomienda usar la función `seq_len()`[7] en detrimento de `seq()` si se desea obtener estos resultados.

> The fifth form generates the sequence 1, 2, ..., length(from) (as if argument along.with had been specified), unless the argument is numeric of length 1 when it is interpreted as 1:from (even for seq(0) for compatibility with S). Using either seq_along or seq_len is much preferred (unless strict S compatibility is essential).

Para más detalle puede revisar la documentación de la función `?seq`.

2.5 Concatenar elementos y vectores.

Para concatenar elementos, como se explicó antes, se usa la función `c()`, asimismo, si deseamos concatenar vectores con elementos tambipen se usará `c()`.

En el siguiente ejemplo se puede observar este procedimiento:

```r
# Se define el vector t2.

t2<-seq(1, 20, by=3)
t2
```

```
## [1]  1  4  7 10 13 16 19
```

Primero definimos el vector `t2`. Ahora, si deseamos agregar el número `0` como primer elemento del vector `t2`, lo haríamos de la siguiente manera.

[7] El ciclo while se desarrollará en el siguiente capítulo.

```
# Agregando el cero al inicio del vector t2.

t2<-c(0,t2)
t2
```

```
## [1]  0  1  4  7 10 13 16 19
```

En efecto, se ha podido agregar el `0` al inicio del vector `t2`. Ahora si deseamos agregar dos elementos y estos que están al final del vector `t2`. Haríamos como sigue.

```
# Si se desea agregar dos elementos al final del vector.
t2<-c(t2, 20, 21)
t2
```

```
##  [1]  0  1  4  7 10 13 16 19 20 21
```

Se ha agregado el `20` y `21` al final del vector `t2`. Entonces, si deseamos agregar elementos a un vector entonces usando la función `c()` podremos conseguir este objetivo.

Como último caso, mostramos como concatenar dos vectores.

```
# Definimos 2 vectores. El vector "x" y "z"
x<-c(1,4,6)
z<-c(3,10,12)
```

Si deseamos concatenarlos tendremos que usar la función `c()`.

```
t3<-c(x,z)
t3
```

```
## [1]  1  4  6  3 10 12
```

En efecto, se han unido los dos vectores y se ha generado el vector `t3` que contiene a `x` y `z`.

3 OPERACIONES MATEMÁTICAS Y ESTADÍSTICAS CON VECTORES.

En este capítulo se desarrollará operaciones matemáticas y estadísticas con vectores. Para este capítulo es fundamental recordar lo que se desarrolló en el capítulo 1 llamado "R como calculadora".

3.1 Operaciones matemáticas.

En el siguiente tabla se mostrarán algunas operaciones que se pueden hacer con vectores en R.

OPERACIONES	SINTAXIS
Adición	+
Sustracción	-
Producto Escalar	%*%
Producto de Elementos	*
Suma de elementos	**sum()**
Producto	**prod()**
Suma Acumulada	**cumsum()**
Producto Acumulado	**cumprod()**
Diferencias	**diff()**

Para mostrar las operaciones primero se definirán 2 vectores que nos permitirán hacer los cálculos.

```
# Creamos el vector x
x<-1:8
x
```

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

```
# Y el vector y
y<-seq(3,27, length.out = 8)
y
```

```
## [1]  3.000000  6.428571  9.857143 13.285714 16.714286 20.142857 23.571429 27.000000
```

Si se toma en cuenta la adición, esta se dará entre los elementos de posiciones iguales. Es decir, se sumará elemento de la primera posición del vector `x` y el elemento de primera posición del vector `y`, y así sucesivamente.

Veamos como sería la suma del vector `x` y `y`.

```
# Adición de vectores.
x+y
```

```
## [1]  4.000000  8.428571 12.857143 17.285714 21.714286 26.142857 30.571429 35.000000
```

Para el caso de la sustracción es similar. Como se muestra en el siguiente ejemplo:

```
# Sustracción de vectores.
x-y
```

```
## [1]  -2.000000  -4.428571  -6.857143  -9.285714 -11.714286 -14.142857 -16.571429
## [8] -19.000000
```

EL producto escalar, como se sabe es la suma agregada del producto de cada elemento de posiciones iguales. Para poder determinarlo se usará la función `%*%`.

```
# Producto escalar.
x%*%y
```

```
##      [,1]
## [1,]  684
```

El producto de elementos se da cuando se multiplica elementos de la misma posición:

```
# Producto de Elementos.
x*y
```

```
## [1]   3.00000  12.85714  29.57143  53.14286  83.57143 120.85714 165.00000 216.00000
```

Si deseamos sumar todos los elementos de un vector se tendrá que usar la función `sum()`. Veamos un ejemplo.

```
# Suma de elementos del vector.
sum(x)
```

```
## [1] 36
```

Con lo visto, otra forma de calcular el producto escalar es usando el producto de elementos y la suma de elementos. Veamos el método alternativo.

```
sum(x*y)
```

```
## [1] 684
```

En efecto, nos resulta lo mismo que si lo hubieras hecho con la función `%*%`.

Ahora que ya conocemos como calcular algunas oeraciones matemáticas, podremos calcular algunas operaciones un poco más complejas como el cálculo de la norma o longitud del vector. Recordar que la norma de vector es: $Norma = \sqrt{x_1^2 + x_2^2 + x_3^2 + ...x_n^2}$

Entonces, este procedimiento podremos hacerlo en R con la siguiente sintaxis.

```
sqrt(sum(x^2))
```

```
## [1] 14.28286
```

En el caso que se desee calcular el producto de todos los elemenos de un vector, se usará la función `prod()`.

```
# El producto de todos los elementos de x.
prod(x)
```

```
## [1] 40320
```

Lo que se ha calculado aquí es la multiplicación de todos los elementos del vector `x`, es decir, 1 por 2 por 3 por 4 por 5 por 6 por 7 y por 8.

Otra operación es la suma acumulada de elementos, para ello se usa la función `cumsum()`, lo que hace es sumar los dos primeros elemento; este resultado lo suma al tercer elemento y así sucesivamente.

```
# Suma acumulada de elementos de x.
cumsum(x)
```

```
## [1]  1  3  6 10 15 21 28 36
```

Como se puede observar, el último elemento de la suma acumulada es igual a la suma de todos los elementos, el resultado que se obtuvo con la función `sum()`.

Similarmente el producto acumulado, se calculará con la función `cumprod()`

```
# El producto acumulado de elementos de x.
cumprod(x)
```

```
## [1]     1     2     6    24   120   720  5040 40320
```

Para diferenciar elemento a elemento se usa la función `diff()`, que realiza la diferencia entre dos elementos contiguos.

```
# Diferencias sucesivas de elementos de x.
diff(x)
```

```
## [1] 1 1 1 1 1 1 1
```

Como el vector `x` es una secuencia que aumenta de 1 en 1. Al aplicar esta función el resultado nos muestra que cada elemento del vector se diferencia en una unidad.

3.2 Operaciones estadísticas.

A continuación se presentarán las funciones para realizar operaciones estadísticas con vectores.

OPERACIÓN	SINTAXIS
Media	**mean()**
Mediana	**median()**
Máximo	**max()**
Mínimo	**min()**
Cuantiles	**quantile()**
Coeficiente de Correlación	**cor()**

Todas las operaciones se harán respecto al vector `x`.

Si se desea calcular la media aritmética o promedio de los elementos del vector, se tendrá que usar la función `mean()`, como sigue:

```
# Cálculo de la media del vector x.
mean(x)
```

```
## [1] 4.5
```

Asimismo, si deseamos calcular la mediana de un vector tenemos que usar la función `median()`. A continuación un ejemplo.

```
# Cálculo de la mediana del vector x.
median(x)
```

```
## [1] 4.5
```

Para hallar el elemento de máximo valor se usará la función `max()`.

```
# El elemento de valor máximo del vector x.
max(x)
```

```
## [1] 8
```

Asimismo, para calcular el elemento de mínimo valor se usará la función `min()`

```
# El elemento de valor mínimo del vector x.
min(x)
```

```
## [1] 1
```

Ahora una de las operaciones más importantes es poder calcular los percentiles de un vector, para esto se usará la función `quantile()`.

```
quantile(x)
```

```
##   0%  25%  50%  75% 100%
## 1.00 2.75 4.50 6.25 8.00
```

R por defecto, nos arroja los resultados para los cuartiles. Es así que podemos ver que se tiene información para el 25%, 50%, 75% y 100%.

Pero si deseamos obtener los quintiles, entonces podemos hacer uso del argumento `probs` de la función `quantile()`. De la siguiente manera.

```
quantile(x, probs = c(0,.2,.4,.6,.8,1))
```

```
##   0%  20%  40%  60%  80% 100%
##  1.0  2.4  3.8  5.2  6.6  8.0
```

En efecto, ahora tenemos información para los quintiles: 20%, 40%, 60%, 80% y 100%.

Entonces, si usted desea calcular los deciles, colocaría en `probs=c(0, .1, .2, .3, .4, .5, .6, .7, .8, .9, 1)`, y si quiere calcular los ventiles la serie sería `0`, `.05`, `.1`, `.15` y así susecivamente hasta el `1`. Usted se habrá podido dar cuenta que esto es muy tedioso y que debe de haber una forma más rápida y sencilla.

En realidad, sí. Usted que ha leído el capítulo 2 sabrá con que función solucionar este pequeño problema. En efecto, es la función `seq()`.

Entonces, si queremos calcular los quintiles usando la función `seq()`, será de la siguiente manera.

```
quantile(x, probs = seq(0,1, by=0.2))
```

```
##   0%  20%  40%  60%  80% 100%
```

```
##  1.0  2.4  3.8  5.2  6.6  8.0
```

Y si desea calcular los deciles y los ventiles.

```
# Para calcular los deciles.
quantile(x, probs = seq(0,1, by=0.1))
```

```
##   0%  10%  20%  30%  40%  50%  60%  70%  80%  90% 100%
##  1.0  1.7  2.4  3.1  3.8  4.5  5.2  5.9  6.6  7.3  8.0
```

```
# Para calcular los ventile.
quantile(x, probs = seq(0,1, by=0.05))
```

```
##   0%   5%  10%  15%  20%  25%  30%  35%  40%  45%  50%  55%  60%  65%  70%  75%  80%
## 1.00 1.35 1.70 2.05 2.40 2.75 3.10 3.45 3.80 4.15 4.50 4.85 5.20 5.55 5.90 6.25 6.60
##  85%  90%  95% 100%
## 6.95 7.30 7.65 8.00
```

En efecto, hemos calculado los deciles y ventiles[8] sin escribir tanto código.

Por último, si deseamos calcular el coeficiente de corrrelación entre dos vectores, usaremos la función `cor()`.

```
# El coeficiente de correlación entre "x" y "y".
cor(x,y)
```

```
## [1] 1
```

La función `cor()` calcula por defecto el coeficiente de correlación de pearson. Pero si usted desea calcularlo con otro método, tendrá que usar el argumento `method`. R nos da la posibilidad de poder calcular el coeficiente de correlación de Kendall y el de spearman, adicionalmente, al de Pearson que viene por defecto.

Es así que si usted desea calcular el coeficiente de correlación de spearman. Tendría que usar la siguiente sintaxis.

```
cor(x,y, method = "spearman")
```

```
## [1] 1
```

3.2.1 ¿Cómo trabajar con missings o elementos faltantes (NA)?

Hasta ahora hemos visto que nuestros datos están completos, pero en el día a día la mayoría de veces no se cuenta con la totalidad de los datos y se tiene **missing values** o mejor conocidos como valores faltantes. En R, a estos valores faltantes se les conoce como `NA` (not available).

Si se tiene `NA` en un vector, entonces, las funciones que hemos utilizado para hacer cálculos matemáticos y estadísticos arrojarán como resultado `NA`. Veamos un ejemplo para que quede claro.

```
# Definimos el vector z.
z<-c(1,4,NA,6,10)
z
```

```
## [1]  1  4 NA  6 10
```

Si deseamos calcular la media del vector `z`.

```
mean(z)
```

```
## [1] NA
```

[8] Si usted no comprende esta sintaxis, le recomendamos revisar el capítulo de vectores aleatorios o buscar cuales son los argumentos de la función `sample()`.

En efecto, nos arrojó como resultado `NA`. Entonces cuando se tenga elementos faltantes en nuestro vector, tendremos que usar el argumento `na.rm`. Este argumento es parte de todas las funciones que hemos desarrollado en este capítulo.

A continuación un ejemplo.

```
mean(x, na.rm = TRUE)
```

```
## [1] 4.5
```

Hemos podido solucionar el problema de los elementos faltantes.

Es así que si usted tiene valores faltantes y desea obtener el cálculo de los estadísticos sin tener en consideración a éstos, tiene que usar el argumento `na.rm=TRUE`.

3.2.2 ¿Cómo ordenar elementos de los vectores?

En este apartado se verá cómo ordenar los elementos de los vectores con las funciones `sort()` y `rev()`[9]. La función `sort()` permite ordenar elementos de manera decreciente o creciente, mientras que la función `rev()` permite obtener el vector en orden inverso.

Veamos los argumentos de la función `sort()`:

```
sort(x, decreasing=FALSE, ...)
```

Donde:
`x`: Es un objeto vector de clase numérico, complejo, caracter o lógico. `decreasing`: Es un argumento lógico en donde si toma el valor de `TRUE` entonces los elementos del vector son ordenados de manera decreciente. Su valor por defecto es `FALSE`. `...`: Otros argumentos.

Para más detalles veamos un ejemplo:

```
# Creamos el vector x.
x<-c(3,5,3,2,1,4,8,9,7,10,13,1,6)
x
```

```
##  [1]  3  5  3  2  1  4  8  9  7 10 13  1  6
```

Podemos ver que el vector `x` que acabamos de crear tiene los elementos desordenados. Así que si queremos ordenarlos de forma creciente, usaremos la función sort.

```
# Ordenando de manera creciente.
sort(x)
```

```
##  [1]  1  1  2  3  3  4  5  6  7  8  9 10 13
```

En efecto, se ha ordenado los elementos del vector `x` de forma creciente. Como puede ver, no hemos usado el argumento `decreasing`, ya que queremos ordenarlo de forma creciente.

Si queremos ordenarlo de forma decreciente, sí tendremos que usaar el argumento `decreasing`

```
sort(x, decreasing = TRUE)
```

```
##  [1] 13 10  9  8  7  6  5  4  3  3  2  1  1
```

El vector `x` ha sido ordenado de forma creciente.

Ahora, si deseamos ordenar los elementos del vector en orden inverso, tendremos que usar la función `rev()`. Veamos su sintaxis.

```
rev(x)
```

[9] El ciclo while se desarrollará en el siguiente capítulo.

Donde:
`x`: Es un vector u otro objeto que pueda ser definido inversamente.

Por ejemplo, si nosotros queremos ordenar el vector x de forma decreciente, una vez que ya se ordenó de manera creciente.

```
# Ordenando el vector x de manera creciente.
x_cre<-sort(x)
x_cre
```

```
##  [1]  1  1  2  3  3  4  5  6  7  8  9 10 13
```

Hemos definido al vector `x_cre` con el vector ordenado crecientemente del vector `x`. Ahora le aplicamos la función `rev()` para que se ordene de forma inversa (decrecientemente).

```
# Ordenando inversamente el vector x_cre
rev(x_cre)
```

```
##  [1] 13 10  9  8  7  6  5  4  3  3  2  1  1
```

En efecto, el vector `x_cre` ha sido ordenado de forma inversa.

4 VECTORES ALEATORIOS y SELECCIÓN DE ELEMENTOS.

En este capítulo usted aprenderá a crear vectores de elementos aleatorios, así también, cómo seleccionar uno o varios elementos de un vector, a crear vectores vacíos y conocer el tamaño de los vectores.

Quizá le parezca muy abstracto lo que se desarrollará en este capítulo, quizá usted se pregunte, ¿en el dia a día cómo voy a trabajar con vectores? o ¿para qué me servirá seleccionar elementos?. Déjeme decirle que en capítulos posteriores, los vectores se convertirán en columnas y las columnas en variables y las variables juntas generarán un dataframe o base de datos. Así que comprender cómo seleccionar elementos o saber el tamaño de vectores le permitirán filtrar observaciones en un futuro.

En este y en el siguiente capítulo está lo más fundamental, si uno desea conocer el funcionamiento y en un futuro empezar a programar en el software R. Es asi que se recomienda leer, estos dos capítulos, las veces que sea posible hasta que quede asimilado por completo.

4.1 Creación de vectores con elementos aleatorios.

Para poder crear vectores con elementos aleatorios tendremos que usar la función `sample()`. Veamos a continuación su sintaxis.

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

Donde:
`x`: Elemento o vector que contiene el universo de elementos.
`size`: Es el tamaño del vector resultante. Es decir, el número de elementos de nuestro vector generado.
`replace`: Argumento lógico que indica si se escogeran los elementos con reemplazo o no. Su valor por defecto es `FALSE`, lo que implica que la elección de elementos sea sin reemplazos.
`prob`: Vector de probabilidades que indica la ocurrencia de los elementos del elemento o vector `x`.

Para comprender su funcionamiento, veamos un ejemplo. Si queremos escoger aleatorimente 5 elementos de un vector con elementos del 1 al 10.

```r
# Escogiendo aleatoriamente 5 elementos.

sample(x=1:10, size = 5)
```

```
## [1] 6 5 1 8 7
```

El vector que contiene el universo de donde se escogeran los 5 números aleatoriamente es `1:10`. Asimismo, con el argumento `size=5` le estamos indicando que escoga 5 elementos. Está escogiendo 5 números distintos, ya que el argumento `replace` se está dejando con su valor por defecto que es `FALSE`, es decir, sin reemplazos.

Es preciso aclarar que usted obtendrá resultados distintos cada vez que ejecute el ejemplo, ya que en cada ejecución el software utiliza un algoritmo distinto para seleccionar elementos. Es decir, está escogiendo aleatoriamente los elementos en cada ejecución.

Veamos como con la misma sintaxis, el software, a escogido distintos elementos.

```r
# Escogiendo aleatoriamente 5 elementos.

sample(x=1:10, size = 5)
```

```
## [1] 10  9  1  4  7
```

Entonces si queremos corregir esto y queremos guardar los resultados y que al volver a correr toda la sintaxis arroje los mismos resultados, tendremos que usar una semilla. En R, colocar una semilla a un proceso aleatorio se realiza usando la función `set.seed()`, que toma como argumento cualquier número (semilla).

Veamos un ejemplo. Vamos a ejecutar la misma sintaxis del ejemplo anterior pero usando una semilla.

```r
set.seed(10)
sample(x=1:10, size = 5)
```

```
## [1] 9 7 8 6 3
```

Cuando usamos la semilla con valor de 10. Nos arroja como resultado los números: `9 8 7 6 y 3`[10].

Veamos como sale el mismo resultado cuando volvemos a ejecutar la misma sintaxis.

```r
set.seed(10)
sample(x=1:10, size = 5)
```

```
## [1] 9 7 8 6 3
```

En efecto, los resultados son los mismos. Con lo cual queda demostrado que si deseamos obtener los mismos resultados de un proceso aleatorio tendremos que usar una semilla.

Ahora veamos un ejemplo un poco más atractivo.

Vamos a seleccionar aleatoriamente 2 personas de un conjunto de 6 personas.

```r
set.seed(12)
sample(x=c("LUIS","MARÍA","JUAN","ROBERTH","CARLOS","FLOR"), size = 2)
```

```
## [1] "MARÍA" "FLOR"
```

El software escogió a `MARÍA Y FLOR`. Este ejemplo se desarrollo con el fin de poder ver un ejemplo en donde el vector universal es un vector de elementos caracter. Ahora veamos un ejemplo usando el argumento `replace`.

Vamos a generar un vector de 20 elementos en donde se escoga entre 1 y 0.

```r
set.seed(20)
sample(x=c(0,1), size = 20, replace = TRUE)
```

[10] Si usted no comprende esta sintaxis, le recomendamos revisar el capítulo de vectores aleatorios o buscar cuales son los argumentos de la función `sample()`.

```
##  [1] 1 0 0 1 1 0 1 0 1 1 0 0 0 0 0 0 0 0 1 1
```

Se ha escogido aleatoriamente 20 elementos que son `1 o 0`. Esto se pudo dar gracias al uso del argumento `replace=TRUE`. Ya que si no se hubiera especificado que haya reemplazos, nos arojaría error ya que no se puede toma una muestra más grande que el vector universal cuandi replace toma el valor de `FALSE`. Veamos este caso a continuación:

```
set.seed(20)
sample(x=c(0,1), size = 20)
```

```
## Error in sample.int(length(x), size, replace, prob): cannot take a sample larger than the population
```

En efecto nos arroja un error, al correr el código.

Ahora veamos un ejemplo en donde se usa el argumento `prob`. Vamos a escoger un vector que tenga 100 elementos entre hombres y mujeres, pero queremos que la proporción de hombres sea del 30% y de mujeres del 70%, aproximadamente.

Primero ejecutemos el problema sin especificar la proporción de hombres y mujeres.

```
set.seed(1805)
sexo<-sample(x=c("HOMBRE", "MUJER"), size = 100, replace = T)
sexo
```

```
##   [1] "HOMBRE" "HOMBRE" "HOMBRE" "HOMBRE" "MUJER"  "MUJER"  "HOMBRE" "HOMBRE" "MUJER"
##  [10] "MUJER"  "HOMBRE" "MUJER"  "MUJER"  "MUJER"  "HOMBRE" "HOMBRE" "MUJER"  "MUJER"
##  [19] "MUJER"  "MUJER"  "MUJER"  "MUJER"  "MUJER"  "HOMBRE" "HOMBRE" "MUJER"  "HOMBRE"
##  [28] "HOMBRE" "MUJER"  "MUJER"  "MUJER"  "HOMBRE" "HOMBRE" "HOMBRE" "HOMBRE" "MUJER"
##  [37] "MUJER"  "MUJER"  "MUJER"  "HOMBRE" "MUJER"  "MUJER"  "HOMBRE" "MUJER"  "MUJER"
##  [46] "MUJER"  "MUJER"  "MUJER"  "MUJER"  "HOMBRE" "HOMBRE" "HOMBRE" "MUJER"  "HOMBRE"
##  [55] "MUJER"  "MUJER"  "HOMBRE" "HOMBRE" "HOMBRE" "HOMBRE" "HOMBRE" "HOMBRE" "MUJER"
##  [64] "MUJER"  "HOMBRE" "MUJER"  "MUJER"  "HOMBRE" "MUJER"  "MUJER"  "MUJER"  "MUJER"
##  [73] "HOMBRE" "HOMBRE" "MUJER"  "HOMBRE" "MUJER"  "MUJER"  "MUJER"  "MUJER"  "MUJER"
##  [82] "HOMBRE" "HOMBRE" "HOMBRE" "HOMBRE" "HOMBRE" "MUJER"  "MUJER"  "HOMBRE" "MUJER"
##  [91] "MUJER"  "MUJER"  "HOMBRE" "HOMBRE" "HOMBRE" "MUJER"  "MUJER"  "HOMBRE" "MUJER"
## [100] "HOMBRE"
```

Estamos guardando los resultados en el vector `sexo`. Si queremos saber cuántos son hombres y cuántos son mujeres usted podría contar uno por uno, pero esto resultaría muy tedioso, la forma más eficiente es ver los resultados en una tabla de valores absolutos. Para esto usamos la función `table()`. Veamos el resultado a continuación.

```
table(sexo)
```

```
## sexo
## HOMBRE  MUJER
##     45     55
```

El software escogió aleatoriamente 45 hombres y 55 mujeres. Para poder obtener estos resultados en valores relativos usamos la función `prop.table()`.

```
prop.table(table(sexo))
```

```
## sexo
## HOMBRE  MUJER
##   0.45   0.55
```

El 45% son hombres y el 55% son mujeres.

El problema en un inicio nos decia generar un vector de hombres y mujeres en donde el 30% sea hombres y el 70% mujeres. Entonces, para poder generar el vector deseado usaremos el argumento `prob`.

```r
set.seed(1805)
sexo<-sample(x=c("HOMBRE", "MUJER"), size = 100, replace = T, prob = c(0.3,0.7))
sexo
```

```
##   [1] "MUJER"  "MUJER"  "MUJER"  "MUJER"  "MUJER"  "MUJER"  "HOMBRE" "HOMBRE" "MUJER"
##  [10] "MUJER"  "HOMBRE" "MUJER"  "MUJER"  "MUJER"  "HOMBRE" "MUJER"  "HOMBRE" "MUJER"
##  [19] "MUJER"  "MUJER"  "MUJER"  "MUJER"  "MUJER"  "MUJER"  "MUJER"  "MUJER"  "MUJER"
##  [28] "MUJER"  "HOMBRE" "MUJER"  "MUJER"  "MUJER"  "MUJER"  "MUJER"  "HOMBRE" "MUJER"
##  [37] "HOMBRE" "HOMBRE" "MUJER"  "MUJER"  "HOMBRE" "MUJER"  "HOMBRE" "MUJER"  "MUJER"
##  [46] "HOMBRE" "MUJER"  "HOMBRE" "HOMBRE" "HOMBRE" "HOMBRE" "MUJER"  "MUJER"  "MUJER"
##  [55] "HOMBRE" "HOMBRE" "HOMBRE" "MUJER"  "MUJER"  "MUJER"  "MUJER"  "MUJER"  "MUJER"
##  [64] "HOMBRE" "MUJER"  "MUJER"  "MUJER"  "MUJER"  "MUJER"  "MUJER"  "MUJER"  "HOMBRE"
##  [73] "MUJER"  "MUJER"  "MUJER"  "MUJER"  "HOMBRE" "MUJER"  "MUJER"  "MUJER"  "MUJER"
##  [82] "HOMBRE" "MUJER"  "HOMBRE" "HOMBRE" "MUJER"  "MUJER"  "HOMBRE" "MUJER"  "MUJER"
##  [91] "MUJER"  "HOMBRE" "MUJER"  "MUJER"  "HOMBRE" "HOMBRE" "MUJER"  "MUJER"  "MUJER"
## [100] "MUJER"
```

Estamos usando la misma semilla, pero estamos especificando las probabilidades[11], además, estamos guardando los resultados en el mismo vector `sexo` (el vector se está reescribiendo). Veamos defrente en valores relativos la proporción de hombres y mujeres.

```r
prop.table(table(sexo))
```

```
## sexo
## HOMBRE  MUJER
##   0.29   0.71
```

El 29% son hombres y el 71% son mujeres. En efecto, gracias a especificar el vector de probabilidades de suceso de cada elemento del vector universal se ha podido obtener la proporción del 30% y 70% entre hombres y mujeres, aproximadamente.

4.1.1 Aplicación: Ley de grandes números.

Una de las leyes más importantes en la teoría de la probabilidad es la "**ley de grandes números**". La cual nos dice que al aumentar el número de ensayos de un experimento, este tiende a la esperanza matemática de la variable aleatoria que se ha generado en el experimento.

Por ejemplo, vamos a definir a un experimento como el lanzamiento de una moneda. Asimismo, definiremos a la variable aleatoria como el número de caras que resulte del experimento. Es así que sí sale cara la variable aleatoria tomará el valor de 1 y si sale sello la variable aleatoria tomará el valor de 0.

Si sacamos la esperanza matemática de este experimento. Nos resultaría 1/2.

$$\text{Probabilidad de que salga cara} = 1/2$$

$$\text{Probabilidad de que salga sello} = 1/2$$

$$\text{Esperanza matemática} = \frac{1}{2} * 1 + \frac{1}{2} * 0 = \frac{1}{2}$$

Lo que implica que el porcentaje de caras será el 50% y el porcentaje de sellos el otro 50%.

Entonces, si nosotros desarrollamos el experimento un número pequeño de veces no nos saldrá necesariamente la mitad de veces cara y la otra mitad sello. Vamos hacer el experimento con 200 ensayos.

[11] El ciclo while se desarrollará en el siguiente capítulo.

```r
set.seed(2020)
experimento<-sample(c("CARA","SELLO"), 200, T)

round(prop.table(table(experimento))*100,4)
```

```
## experimento
##  CARA SELLO
##  46.5  53.5
```

Vemos que el porcentaje de caras es del 46.5% y el de sellos es de 53.5%. Pero por la ley de grandes números nosotros sabemos que al aumentar el número de ensayos el experimento tenderá a la esperanza matemática, es decir, 50% de caras y 50% de sellos.

Veamos el resultado cuando el número de ensayos es de 1000.

```r
set.seed(2020)
experimento<-sample(c("CARA","SELLO"), 1000, T)

round(prop.table(table(experimento))*100,4)
```

```
## experimento
##  CARA SELLO
##  48.4  51.6
```

Con 1000 ensayos el porcentaje de caras es del 48.4% y el de sellos es de 51.6%. Se va acercando a la esperanza matemática. Veamos que resulta con 100000 de ensayos.

```r
set.seed(2020)
experimento<-sample(c("CARA","SELLO"), 100000, T)

round(prop.table(table(experimento))*100,4)
```

```
## experimento
##   CARA  SELLO
## 50.136 49.864
```

Con 100000 ensayos el porcentaje de caras es del 50.14% y el de sellos es de 49.86%. Ya casi es igual al valor de la esperanza matemática.

Por último, veamos que pasa si consideramos 1000000 ensayos.

```r
set.seed(2020)
experimento<-sample(c("CARA","SELLO"), 1000000, T)

round(prop.table(table(experimento))*100,4)
```

```
## experimento
##    CARA   SELLO
## 50.0036 49.9964
```

Con 1000000 ensayos el porcentaje de caras es del 50% y el de sellos es de 50%. Es así como hemos podido demostrar la ley de grandes números.

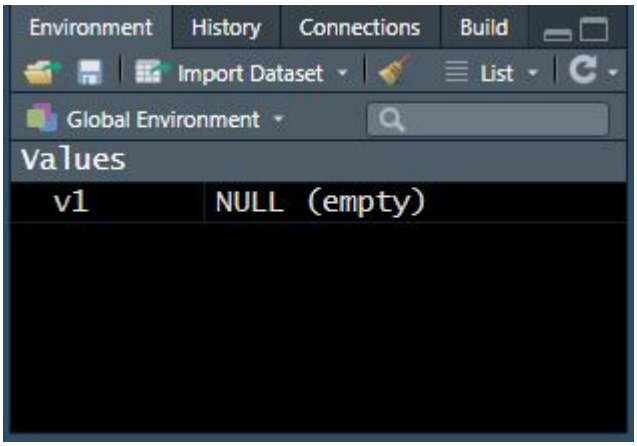
4.2 Vectores vacios y selección de elementos.

4.2.1 Vectores vacíos.

Hasta el momento hemos visto que un vector puede tener uno o más elementos. Pero también están los vectores que no tienen elementos y que son vacíos. A continuación veamos un ejemplo de un vector vacío.

```
# Vector vacio.
v1<-c()
```

Hemos creado el vector vacío, en el environment podemos ver que el vector `v1` está vacío, pero que existe en memoría.



Estos vectores nos serán útiles cuando veamos el tema de ciclos y el tema de funciones. Por el momento es suficiente con saber de su existencia.

4.2.2 Seleción de elementos usando [].

Si nosotros necesitamos selecionar un elemento o un conjunto de elementos de un vector, tendremos que usar el operador `[]`. Veamos unos ejemplos.

```
# Creando un vector.
x<-c(10,3,65,3,2,5,7,6,5,9,99,100,102,1,2,2,3,6,87,12,12,13)
x
```

```
##  [1]  10   3  65   3   2   5   7   6   5   9  99 100 102   1   2   2   3   6  87  12  12
## [22]  13
```

Hemos creado el vector `x`. Si usted quiere seleccionar el elemento de posición 8, usted debería de hacer lo siguiente.

```
# Seleccionado el elemento de posición 8.
x[8]
```

```
## [1] 6
```

En efecto, nos ha selecionado el elemento 6 que corresponde al elemento de posición 8 contando de izquierda a derecha. Entonces, si queremos seleccionar o extraer elementos de un vector, primero se tiene que indicar el vector del cual se desea extraer el elemento o elementos luego abrir corchetes y colocar la posición del elementos y cerrar los corchetes.

Veamos otro ejemplo en donde se selecciona el elemento de posición 5.

```
# Seleccionado el elemento de posición 5.
x[5]
```

```
## [1] 2
```

Nos resulta el 2, si usted cuenta de izquierda a derecha el elemento de posición 5 es el elemento 2.

Hasta ahora hemos visto como extraer sólo un elemento, pero podemos seleccionar varios elementos, para esto usaremos vectores en vez de elementos dentro de los corchetes. A continuación un ejemplo.

```
# Seleccionado los elementos de posición 3 al 5.
x[3:5]
```

```
## [1] 65  3  2
```

En efecto, hemos seleccionado los elementos de posición 3, 4 y 5 que son 65, 3, 2. Entonces, si deseamos seleccionar varios elementos, usaremos la posición de estos, pero definido como un vector. Veamos un ejemplo adicional en donde se selecciona los elementos de posición 7, 12 y 4.

```
# Seleccionado los elementos de posición 7, 12 y 4.
x[c(7,12,4)]
```

```
## [1]   7 100   3
```

Hemos definido el vector de posiciones deseadas con `c(7,12,4)`. Y nos ha resultado los elementos 7, 100, 3. Con esta especificación se espera que se haya comprendido claramente el uso de vectores dentro de los corchetes.

Pero si nosotros queremos seleccionar un conjunto de elementos pero indicando exclusión, es decir, indicando que no seleccione a uno o varios elementos, entonces usaremos el operador `-`. Con esto le indicaremos que nos seleccione todo el vector a excepción de (`-`). Veamos un ejemplo, en donde se desea seleccionar a todo el vector `x`, pero a excepción del primer elemento.

```
# Seleccionado todo el vector x, a excepción, del elemento de posición 1.
x[-1]
```

```
##  [1]   3  65   3   2   5   7   6   5   9  99 100 102   1   2   2   3   6  87  12  12  13
```

En efecto, se ha selecionado todos los elemento del vector `x`, pero sin considerar el primer elemento. Esta especifiación lo podemos proyectar para excluir a varios elementos. Veamos un ejemplo en donde se quiere excluir a los elementos de posición 6, 10 y 3.

```
# Seleccionar el vector x, a excepción, de los elementos de
# posición 6, 10 y 3.
x[-c(6,10,3)]
```

```
##  [1]  10   3   3   2   7   6   5  99 100 102   1   2   2   3   6  87  12  12  13
```

El resultado es el deseado.

4.2.3 ¿Cómo saber cuántos elementos tiene un vector?

A veces, el vector a estudiar es un vector muy grande y no conocemos su dimensión (total de elementos). Entonces, para conocer el número de elementos de un vector usaremos la función `length()`, que tiene sólo un argumento que es el objeto del cual se desea saber su dimensión. Veamos un ejemplo y conozcamos cuántos elementos tiene el vector `x`.

```
# Para saber el número de elementos del vector x.
length(x)
```

```
## [1] 22
```

El software nos indica que el número de elementos del vector `x` es de 22.

Asimismo, podemos usar esta función para seleccionar elementos (uniendo con lo visto anteriormente). Es así que si deseamos seleccionar el último elemento del vector `x`, podemos usar la siguiente sintaxis.

```
# Seleccionado el último elemento del vector x.
x[length(x)]
```

```
## [1] 13
```

En efecto, hemos seleccionado el último elemento del vector `x`. Por último, si queremos seleccionar los últimos 4 elementos.

```
# Seleccionado los últimos 4 elementos.
x[(length(x)-3):length(x)]
```

```
## [1] 87 12 12 13
```

Recordar que si deseamos seleccionar varios elementos, lo que tiene que estar dentro de los corchetes tiene que ser un vector, por eso colocamos `(length(x)-3):length(x)`. La primera parte `(length(x)-3)` hace referencia a inicio, es decir, desde que posición se desea seleccionar (22-3 = 19); la segunda parte `length(x)` indica el final (22). Es así que seleccionará desde el elemento 19 al 22 (los 4 últimos elementos).

Usted puede pensar que es tedioso y podría ser más eficiente definir la sintaxis con los números, de esta manera `x[19:22]`. Déjeme decirle, que tiene cierta razón, pero su opción sólo sería viable cuando conozca el número total de elementos y tenga un sólo vector. Cuando usted desee automatizar procesos y trabaje con muchos vectores y de una dimensión colosal (que es lo que día a día se encuentra en un centro de trabajo). Usted verá la importancia y le parecerá más factible la sintaxis que hemos usado. Por el momento, estamos realizando ejemplos muy básicos, en capítulos posteriores verá la importancia de este tipo de sintaxis.

5 OPERADORES LÓGICOS, ÍNDICE DE ELEMENTOS Y CAMBIO DE VALORES.

En este capítulo se tratará de cómo usar operados lógicos como: `y`, `o`, `igualdad`, `mayor que`, `menor que`, `diferente`, `negación`, entre otros. Gracias a estos operadores lógicos podremos filtrar y seleccionar elementos y variables de bases de datos. Además se aprenderá como encontrar el índice de elementos y cambiar valores de estos elementos siempre y cuando cumplan una condición.

5.1 Operadores lógicos.

Al igual que las operaciones aritméticas, las operaciones lógicas tienen una importancia primordial ya que a través de ellas se pueden formar condicionales que nos simplificarán el procesamiento de datos.

Al usar un operador lógico en R, es como si le preguntaramos al software y él se limitara a responder sólo con un `TRUE` o `FALSE`. El programa no te arrojará otras respuestas a menos que hayas especificado mal la sintaxis para el uso de operadores lógicos. Asimismo, para R el valor numérico de `TRUE` es 1 y para `FALSE` es 0, esto nos permitirá, por ejemplo, saber cuántos elementos de un vector cumplen lo especificado en la operación lógica y cuántos no.

En la siguiente tabla se muestra los operadores lógicos más importantes:

OPERADOR LÓGICO	SINTAXIS
Igualdad	==
Mayor que	>
Menor que	<

OPERADOR LÓGICO	SINTAXIS
Mayor igual que	>=
Menor igual que	<=
Diferente	!=
Negación	!
y	&
o	|
Pertenece	%in%

Primero definiremos un vector del cual podremos desarollar ejemplos usando los operadores lógicos.

```
# Definiendo el vector x.
x<-c(5,2,4,8,6,4,0,8,1,4,7,3,11,5,1,1,2,3,4,3,3,7,7,10)

# Mostrando el vector x.
x
```

```
##  [1]  5  2  4  8  6  4  0  8  1  4  7  3 11  5  1  1  2  3  4  3  3  7  7 10
```

5.1.1 Igualdad.

Si nosotros queremos comprobar si dos objetos son iguales, entonces usaremos el operador de igualdad `==`. Es así que si deseamos saber si dos o más elementos son iguales o si uno o más vectores son iguales o otras operaciones tendremos que usar este operador. Veamos un ejemplo muy sencillo para comprenderlo.

Para esto vamos a consultar a **R** si 5 es igual a 7.

```
# ¿El número 5 es igual al número 7?

5==7
```

```
## [1] FALSE
```

Como se muestra **R** nos arroja el booleano FALSE. El cual nos indica que `5` no es igual a `7`. Ahora veamos un ejemplo en donde dos elementos son iguales.

```
# ¿El número 2.4 es igual a 2.4?
2.4==2.4
```

```
## [1] TRUE
```

Como era obvio, nos arroja el booleano TRUE.

Hasta ahora hemos visto como saber si un elemento es igual a otro. Pero esta operción se puede proyectar a vectores. Por ejemplo, vamos a consultar a **R** que elementos del vector `x` son iguales a `3`.

```
# ¿Qué elementos del vector x son iguales a 3?
x==3
```

```
##  [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE
## [15] FALSE FALSE FALSE  TRUE FALSE  TRUE  TRUE FALSE FALSE FALSE
```

Podemos ver que nos arroja `TRUE` o `FALSE` por cada elemento del vector `x`. Nos arroja `TRUE` en el elemento de posición 12, lo que nos indica que el elemento de posición 12 es igual a `3`. Asimismo, también encontramos otros `TRUE` en los elementos de posición 18, 20, 21, lo que nos indica también que en esos elementos del vector `x` son iguales a `3`.

Veamos que sucede si preguntamos a **R** que elementos del vector `x` son iguales a `33`.

```r
# ¿Qué elementos del vector x son iguales a 33?
x==33
```

```
##  [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [15] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
```

El resultado que obtenemos son todos `FALSE`, lo que nos indica que no hay un elemento en el vector `x` que sea igual a `33`.

Hagamos una comprobación que nos arroje un similar resultado. Para esto vamos a decirle a R que sume cuantos elementos del vector `x` son iguales a `33`. Entonces, con la siguiente sintaxis sé que el total de elementos de `x` no son iguales a `33`.

```r
# Método que nos permite comprobar los resultados.
sum(x==33)==0
```

```
## [1] TRUE
```

Lo que hemos hecho es sumar (numéricamente los `TRUE` y `FALSE`), recordar que para R `TRUE` significa 1 y `FALSE` signfica 0. Entonces, si sumamos 33 `FALSE` nos resultará `0` que es igual al `0` que colocamos en la parte derecha del operador lógico.

Y en efecto, el resultado es TRUE, lo que nos indica que no hay ningún elemento en el vector `x` que sea igual a `33`.

5.1.2 Mayor que.

Si queremos saber si un elemento es "mayor que" otro elemento, entonces, se tiene que usar el operador lògico `>`. Este operador lógico también se puede proyectar a más objetos. Veamos un ejemplo, vamos a consultar a R si, 5 es mayor que 3.

```r
# ¿El número 5 es mayor que 3?
5>3
```

```
## [1] TRUE
```

Como era obvio, nos arroja el valor de TRUE, ya que, 5 es mayor que 3. Ahora veamos un ejemplo usando vectores. Vamos a consultarle a R, qué números son mayores que 8.

```r
# ¿Qué elementos del vector x son mayores que 8?
x>8
```

```
##  [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE
## [15] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE
```

Podemos observar que los elementos de posición son el 13, 24.

5.1.3 Menor que.

Si queremos saber si un elemento es "menor que" otro elemento, entonces, se tiene que usar el operador lògico `<`. Este operador lógico también se puede proyectar a más objetos como: vectores, matrices, listas, etc. Veamos un ejemplo, vamos a consultar a R si, 7 es menor que 10.

```r
# ¿El número 7 es menor que 10?
7<10
```

```
## [1] TRUE
```

En efecto, el resultado es verdadero `TRUE`, ya que, 7 es menor que 10. Ahora veamos un ejemplo en donde se le consunlta a R que elementos del vector `x` son menores que 3.

```
# ¿Qué elementos del vector x son menores que 3?
x<3
```

```
##  [1] FALSE  TRUE FALSE FALSE FALSE FALSE  TRUE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE
## [15]  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
```

Nos muestra que la posición de los elementos menores que 3 son: 2, 7, 9, 15, 16, 17.

5.1.4 Diferente.

Otro de los operadores lógicos de mucha relevancia es el operador "diferente" que en R se representa con `!=`. Este operador servirá cuando queremos saber si un número es diferente a otro o cuando deseamos filtrar observaciones que son distintas al valor evaluado. Veamos un ejemplo en donde consultamos a R, si el 8 es diferente del 9.

```
# ¿El número 8 es diferente del 9?
8!=9
```

```
## [1] TRUE
```

Como es obvio nos arroja TRUE, ya que, el 8 es diferente del 9. Veamos un ejemplo usando vectores. Vamos a consultarle a R que nos arroje los elementos que son distintos de 7.

```
# ¿Qué elementos del vector x que son diferentes de 7?
x!=7
```

```
##  [1]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE  TRUE  TRUE  TRUE
## [15]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE FALSE  TRUE
```

Nos arroja muchos `TRUE`, por lo que la gran mayoría de elementos son distintos de 7. Veamos cuántos elementos son diferentes de 7.

```
# ¿Cúantos números cumplen la condición?
sum(x!=7)
```

```
## [1] 21
```

R nos dice que hay 21 elementos que son diferentes de 7.

5.1.5 Negación.

La negación sirve para mutar el booleano de `TRUE` a `FALSE` o de `FALSE` a `TRUE`. Es así que si deseo consultar a R por una cuestión, puedo negar esta cuestión y obtener el resultado contrario. Veamos un ejemplo, en donde consultamos a R si 10 es igual a 8. Obviamente, nos dirá que es `FALSE`, pero si usamos la negación el booleano mutará a `TRUE`.

```
# ¿El número 10 es igual 8?
10==8
```

```
## [1] FALSE
```

```
# Usando la negación.
!(10==8) # ¿El número 10 es diferente del número 8?
```

```
## [1] TRUE
```

En efecto, al negar la consulta hemos obtenido el resultado contrario. Ahora veamos un ejemplo usando vectores. Vamos a consultarle a R cuales son los elementos que son iguales a 5, pero lo negaremos, por lo cual nos arrojará los que son distintos de 5.

```r
# ¿Qué elementos del vector x son iguales a 5?
x==5
```

```
##  [1]  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE
## [15] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
```

```r
# Usando la negación.
!(x==5)  # Elementos distintos de 5.
```

```
##  [1] FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE
## [15]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
```

El resultado es muy intuitivo y reulta ser el complemento del primero. Este tipo de operadores lógicos se usará con mayor frecuencia cuando se haga operaciones lógicas usando caracteres.

5.1.6 Y condicional (Conjunción).

El `Y` condicional es un operador lógico que representa la intersección entre dos o más operaciones lógicas. Es así que si deseamos obtener el resultado cuando si o si se cumplan dos o más operaciones lógicas, entonces, usaremos la condicional "y" que se representa como `&`.

veamos un ejemplo, en donde se le consulta a R si el número 3 es menor que 8 y que el caracter "azul" es diferente del caracter "rojo".

```r
# ¿El número 3 es menor que 8 y el caracter azul
# es diferente del caracter rojo?
3<8 & "azul"!="rojo"
```

```
## [1] TRUE
```

Como era de esperarse el resultado es TRUE, ya que 3 es menor que 8 `y` además azul es distito de rojo.

Ahora veamos un ejemplo en donde se usará más de dos operaciones lógicas.

Se el consultará a R si el 5 es mayor que 4 y si el 10 es menor que 6 y si 8 es igual a 8.0.

```r
# ¿el 5 es mayor que 4 y si el 10 es menor que 6
# y si 8 es igual a 8.0?

5>4 & 10<6 & 8==8.0
```

```
## [1] FALSE
```

El resultado que obtenemos es FALSE, ya que la primera operación es verdadera, la segunda falsa y la tercera verdadera. Lo que en conjunto resulta ser FALSE.

Pero usted, puede estar preguntandose por qué un verdadero, un falso y otro verdadero da falso. Esto se puede entender de dos formas. La primera, intuitivamente; y la segunda, con la tabla de verdades de los cursos de lógica proposicional que se imparten en la educación básica. Veamos a detalle cada una de estas formas.

La forma intuitiva, si nosotros queremos saber el resultado de un conjunto de operaciones lógicas, nosotros sabemos que el resultado podría ser `TRUE` o `FALSE`. El operador `y (&)` representa a la conjunción, ésta une dos o más proposiciones con el fin de poder construir a un proposición mayor y si esta intersección de proposiciones es verdadera entonces el resultado será verdadero, ya que estas dos proposiciones se podrán intercambiar. Entonces, será verdadero cuando el conjunto de operaciones lógicas resultan verdadero, ya que una verdad y otra verdad y otra verdad y otra verdad serán `VERDAD`. Si una de ellas es falsa, entonces el conjunto de operaciones lógicas resultará falso, ya que se rompe la relación de igualdad entre las operaciones lógicas.

La segunda forma, usando la tabla de verdades de los cursos de lógica proposicional, estoy seguro que usted recordará de sus cursos de lógica la siguiente tabla.

P	Q	P ^ Q
V	V	**V**
V	F	**F**
F	V	**F**
F	F	**F**

En este cuadro podemos ver dos proposiciones la proposición **P** y la proposición **Q**, cada una de estas proposiciones puede tomar el valor de `V (verdad)` o `F (falso)`, esto se representa en las dos primeras columnas de la tabla anterior. En la última columna tenemos el resultado de la intersección de **P** y **Q**, en donde observamos que sólo dos verdades resultan en verdad, en los otros casos resulta falso. Es así que si nosotros deseamos ejecutar un proceso y si o si el conjunto de proposiciones que evaluamos tienen que ser verdaderas entonces usaremos el operador `&`.

5.1.7 O condicional (disyunción).

La `O` condicional representa la unión entre una y más operaciones lógicas, en R se representa con `|`. Veamos un pequeño ejemplo, en donde consultamos a R si el 8 es igual a 9 o el 10 es igual 10.0.

```
# ¿El número 8 es igual al número 9 o el
# número 10 es igual al número 10.0?

8==9 | 10==10.0
```

```
## [1] TRUE
```

El resultado que obtenemos es TRUE, ya que la condicional `o` arroja la unión de las dos proposiciones. Es así que si la primera es `FALSE` y la segunda `TRUE`, entonces, el resultado es `TRUE`, ya que una de las proposiciones resultó ser `TRUE`.

A continuación mostramos la tabla de verdades para un mejor entendimiento.

P	Q	P v Q
V	V	**V**
V	F	**V**
F	V	**V**
F	F	**F**

Sólo si el resultado en las dos proposiciones es falso entonces el resultado será falso, en los otros casos el resultado será verdadero. Es así que si nosotros queremos ejecutar un proceso y sólo queremos que una se las proposiciones sea verdadera para poder ejecutarlo, entonces usaremos el operador `|`.

5.1.8 Mayor y menor igual que.

La diferencia con los operadores anteriores (mayor que y menor que) es que ahora se considerarán como límites los valores iguales al número evaluado. Veamos dos ejemplos, en el primero se consultará a R si el 4 es mayor igual a 18 y el 9 es menor igual que 9,

```
# ¿El número 4 es mayor igual que 18?
4>=18
```

```
## [1] FALSE
```

```
# ¿El número 9 es menor igual que 9?
9<=9
```

```
## [1] TRUE
```

En el primer caso, el 4 no es mayor igual que 18; y en el segundo, el 8 si es menor igual que 9.

Proyectándolo a vectores. Vamos a er el ejemplo en donde se le consulta a R la posición de elementos que son mayores igual a 5 y menores iguales que 8.

```
# ¿Qué elementos del vector x son mayores iguales que 5
# y menores iguales que 8?

x>=5 & x<=8
```

```
##  [1]  TRUE FALSE FALSE  TRUE  TRUE FALSE FALSE  TRUE FALSE FALSE  TRUE FALSE FALSE  TRUE
## [15] FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE FALSE
```

Los elementos de posición 1, 4, 5, 8, 11, 14, 22, 23 son mayores iguales que 5 y menores iguales que 7. Adicionalmete, si queremos saber cuántos números son, usamos la siguiente sintaxis.

```
# Para saber cuántos números cumplen la condición.
sum(x>=5 & x<=8)
```

```
## [1] 8
```

R nos indica que hay un total de 8 números que cumplen la condición. Es probable que usted piense que esto es en vano, ya que podría contar el número de `TRUE` sin la necesidad de correr `sum(x>=5 & x<=8)`. Esto sucede porque nuestro ejemplo es pequeño, pero si trabaja con datos reales, el tamaño de observaciones es mucho mayor. Así que esta última parte del codigo servirá mucho.

5.1.9 Operador pertenece.

El operador pertenece juega el mismo papel que en teoría de conjutos. Es así que si queremos saber si uno o varios elementos pertenecen a un objeto usaremos este operador. En R este operador se representa como `%in%`. El operador pertenece se usa mucho cuando se trabaja con caracteres.

Veamos un ejemplo en donde vamos a consultarle a R si el elemento 8 pertence al vector x.

```
# Definimos el vector x.
x<-c(1:10,2,3,4,3)

# ¿El número 8 pertence al vector x?
8 %in% x
```

```
## [1] TRUE
```

Como era de esperarse nos arroja el valor de TRUE. Ahora veamos un ejemplo usando caracteres.

Vamos a consultarle a R si los elementos “Lunes” y “Martes” pertenecen al vector x.

```
# Definimos el vector x.
x<-c("Lunes","Martes","Miércoles","Jueves","Viernes","Sábado","Domingo")

# ¿Los elementos "Lunes" y "Martes" pertenecen al vector x?

c("Lunes","Martes") %in% x
```

```
## [1] TRUE TRUE
```

En efecto, nos arroja como resultado TRUE, TRUE.

¿Pero que sucede si invierto el orden de los objetos?

El resultado que obtenemos será como si preguntaramos lo siguiente: los elementos del vector x pertenecen al objeto `c("Lunes","Martes")`.

```
x %in% c("Lunes","Martes")
```

```
## [1]  TRUE  TRUE FALSE FALSE FALSE FALSE FALSE
```

Nos arroja `TRUE`, `TRUE` y luego sólo `FALSE`. Esto es porque el primer elemento del vector x está en el objeto, lo mismo para el segundo elemento del vector x. Pero el tercer elemento del vector x ("Miércoles") no está en el objeto.

El operador pertenece sirve muchísimo porque nos ayuda a simplificar mucho la sintaxis de grandes operaciones lógicas. Veamos un ejemplo.

Le preguntamos a R que elementos del vector x son iguales a "A" o "B" o "C".

```
# Definimos el vector x.
set.seed(20)
x<-sample(c("A","B","C","D"),20,T)
x
```

```
##  [1] "B" "C" "C" "D" "B" "A" "B" "A" "B" "B" "A" "C" "C" "A" "C" "A" "A" "A" "B" "D"
```

```
# ¿Qué elementos del vector x son iguales a "A" o "B" o "C"?
x=="A" | x=="B" | x=="C"
```

```
##  [1]  TRUE  TRUE  TRUE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
## [15]  TRUE  TRUE  TRUE  TRUE  TRUE FALSE
```

Pero si usamos el operador pertenece (`%in%`) la sintaxis sería más limpia y más corta.

```
# La misma solución pero usando el operador pertenece.

x %in% c("A","B","C")
```

```
##  [1]  TRUE  TRUE  TRUE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
## [15]  TRUE  TRUE  TRUE  TRUE  TRUE FALSE
```

5.2 Encontrar el índice de elementos y cambiar valores.

En este capítulo hemos visto hasta el momento que se puede determinar que elemento(s) pertenecen al vector (siempre y cuando se cumplan determinada operación lógica), pero si uno desea saber la posición de este(os) elemento(s). Se tendrá que usar la función `which()`. Esta función permite encontrar la posición de los elementos siempre y cuando se cumpla una determinada operación lógica.

Veamos los argumentos de la función `which()`.

```
which(x, arr.ind = FALSE, useNames = TRUE)
```

Donde:

`x` = Es un vector lógico o un array. `arr.ind` = Argumento lógico que se usa cuando `x` es un array. Si es TRUE, entonces, devuelve los índices en una matriz cuando x es una matriz. `ùseNames` = Argumento lógico que se usa cuando `x` es una array. Si es TRUE, entonces, los nombres del array se deben de mostrar.

Para este capítulo solo usaremos la función `which()` cuando el argumento `x` es lógico, así que no se tendrá en cuenta los dos últimos argumentos, ya que estos se usan cuando `x` es una array.

Entonces, vamos a ver un ejemplo. Pero primero vamos a crear el vector `x` que tiene los sigientes elementos.

```r
# Creando el vector x.
set.seed(2021)
x<-sample(1:100,50,T)
```

Se crea un vector aleatorio de 50 elementos los cuales pueden ser entre el 1 al 100 y pueden repetirse.

Veamos los elementos del vector `x` que acabamos de crear.

```r
x
```

```
##  [1]  7 38 46 58 12 70 64 38 69 23 76 51 60 18 60  3 46 26 36 86 31 98 19 68 86  5 73 63
## [29] 38 18 43 59 70 86 70 63 79 34 22 16 79 88 17 20 68 94 72 83 67  2
```

Nosotros queremos saber las posiciones de los elementos del vector `x` son mayores que 50. Para esto realizamos la siguiente sintaxis.

```r
# Para encontrar la posición de los elementos que son
# mayores a 50 del vector x.

which(x>50)
```

```
##  [1]  4  6  7  9 11 12 13 15 20 22 24 25 27 28 32 33 34 35 36 37 41 42 45 46 47 48 49
```

Lo que nos ha arrojado R son las posiciones de los elementos que son mayores a 50 del vector `x`. Es así que el elemento de posición 4 es mayor a 50. Si nos fijamos en el vector x, el elemento de posición 4 es el elemento de valor 58 que es mayor a 50.

5.2.1 Cambiar valores de elementos.

Una práctica común es cambiar el valor de los elementos de una variable siempre y cuando cumpla una condición o si queremos cambiar el valor de un elemento determinado.

Veamos un ejemplo, para esto vamos a crear el vector aleatorio z.

```r
# Creando el vector aleatorio z.
set.seed(2021)
z<-sample(1:10, 30, T)
```

El vector `z` es un vector aleatorio de 30 elementos que pueden ser entre el 1 y 10 donde los elementos se repiten. El vector `z` tiene los siguientes elementos.

```r
z
```

```
##  [1]  7  6 10  7  4  6  6  6  5  7  9  7  3  2  3  8 10  4  5  6  2  3  4  5  6  5  9  6
## [29]  2  2
```

Por ejemplo. Si se desea cambiar el elemento de posición 9 por el 20, tendríamos que realizar la siguiente sintaxis.

```r
# Cambiando el valor del elemento de índice 9
# por el valor de 20 en el vector z.
z[9]<-20
```

Veamos como ha quedado el vector `z`.

```r
z
```

```
##  [1]  7  6 10  7  4  6  6  6 20  7  9  7  3  2  3  8 10  4  5  6  2  3  4  5  6  5  9  6
## [29]  2  2
```

En efecto, lo que hemos hecho es cambiar el valor del elemento de índice 9 por el valor de 20.

Ahora veamos un ejemplo en donde se desea cambiar por 100 todos los elementos que son iguales a 3. La sintaxis sería la siguiente.

```
z[z==3]<-100
```

Veamos como nos ha quedado el vector `z`.

```
z
```

```
##  [1]   7   6  10   7   4   6   6   6  20   7   9   7 100   2 100   8  10   4   5   6   2
## [22] 100   4   5   6   5   9   6   2   2
```

En efecto, todos los elementos que valián 3 ahora valen 100.

Veamos un último ejemplo en donde los elementos de valor 5 se le sume 1000 con lo cual se convertirán en 1005. Para desarrollar lo siguiente usaremos la siguiente sintaxis.

```
z[z==5]<-z[z==5]+1000
```

Y en efecto, los elementos de valor 5 ahora valen 1005.

```
z
```

```
##  [1]    7    6   10    7    4    6    6    6   20    7    9    7  100    2  100    8   10
## [18]    4 1005    6    2  100    4 1005    6 1005    9    6    2    2
```

En el siguiente capítulo se verán los temas de factores y listas, objetos muy importantes en la programación en el software R.

6 FACTORES Y LISTAS.

En este capítulo se desarrollará el tema de factores donde los niveles no tienen un orden y otro en donde los niveles si están ordenados, para esto se usará las funciones: `factor()` y `ordered()`. Asimismo, se crearán listas y explicaremos para que sirven y cuando usarlas.

6.1 Factores.

En **R** un objeto factor es una variable que asigna índices a las categorías de las variable. Esto permite una mayor facilidad de procesamiento que si se mantendría la variable en string. Por otro lado, también sirve para asignar categorías a números, este uso es característico de softwares como STATA y SPSS. Hay dos tipos de variables factor en **R**: los que tienen los niveles desordenados (porque no es necesario que haya un orden) y los que tienen niveles ordenados. En el primero caso usaremos la función `factor()` o su variante `as.factor()`, y para el segundo caso, usaremos la función `ordered()`.

6.1.1 Factores con niveles desordenados.

Los niveles no necesitarán un orden cuando las categorías de las variables no tienen un orden jerárquico. Por ejemplo, la variable sexo que tiene dos categorías: masculino y femenino. En esta variable el género másculino no vale más que el género femenino ni viceversa. Otro ejemplo, es el estado civil, en esta variable las categorías no tienen un orden jerarquico, ya que, ser soltero no es más o menos que estar casado o estar viudo.

Una vez que se ha comprendido cuando las categorías no tienen un orden jerarquico, podemos ver unos ejemplos para ver como funciona en el software la función `factor()`.

Primero vamos a crear un vector aleatorio llamado `sexo` en donde se tienen las dos categorías: masculino y femenino.

```r
# Generando el vector sexo.
set.seed(2021)
sexo<-sample(c("MASCULINO", "FEMENINO"), 33, T)

# Hemos creado al vector sexo con 33 elementos.
sexo
```

```
##  [1] "MASCULINO" "FEMENINO"  "FEMENINO"  "FEMENINO"  "MASCULINO" "FEMENINO"  "FEMENINO"
##  [8] "FEMENINO"  "FEMENINO"  "FEMENINO"  "MASCULINO" "FEMENINO"  "FEMENINO"  "MASCULINO"
## [15] "MASCULINO" "MASCULINO" "MASCULINO" "MASCULINO" "FEMENINO"  "MASCULINO" "FEMENINO"
## [22] "FEMENINO"  "MASCULINO" "FEMENINO"  "MASCULINO" "MASCULINO" "FEMENINO"  "FEMENINO"
## [29] "FEMENINO"  "FEMENINO"  "MASCULINO" "FEMENINO"  "MASCULINO"
```

Podemos darnos cuenta que todos los elementos del vector `sexo` son character. Entonces, si nosotros queremos colocar un índice a cada variable y así mejorar el procesamiento, entonces necesitamos convertir a factor la variable sexo.

La forma más sencilla es usando la función `as.factor()`. Veamos como:

```r
# Convirtiendo a factor la variable sexo.
sexo1<-as.factor(sexo)
```

Estamos guardando el resultado en la variable `sexo1`.

```r
sexo1
```

```
##  [1] MASCULINO FEMENINO  FEMENINO  FEMENINO  MASCULINO FEMENINO  FEMENINO  FEMENINO
##  [9] FEMENINO  FEMENINO  MASCULINO FEMENINO  FEMENINO  MASCULINO MASCULINO MASCULINO
## [17] MASCULINO MASCULINO FEMENINO  MASCULINO FEMENINO  FEMENINO  MASCULINO FEMENINO
## [25] MASCULINO MASCULINO FEMENINO  FEMENINO  FEMENINO  FEMENINO  MASCULINO FEMENINO
## [33] MASCULINO
## Levels: FEMENINO MASCULINO
```

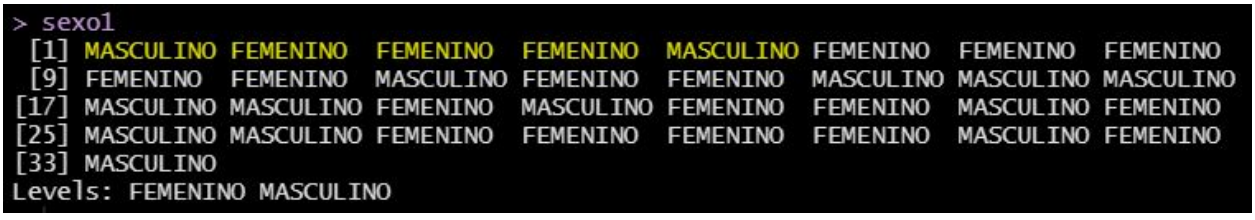
Ahora los elementos del vector no son character, además, en la parte inferior se observa que se tiene dos niveles: `FEMENINO Y MASCULINO`. Lo que ha hecho el software es asignar un índice a cada categoría de la variable `sexo`, en este caso ha asignado el índice 1 a la categoría femenino y 2 a la categoría masculino. Para observar esto usamos la función `str()` que nos arroja la estructura de cada vector.

```r
# Viendo la estructura de la variable sexo1.
str(sexo1)
```

```
##  Factor w/ 2 levels "FEMENINO","MASCULINO": 2 1 1 1 2 1 1 1 1 1 ...
```

En efecto, nos arroja que la variable `sexo1` tiene dos niveles: femenino y masculino y que el índice del primero es 1 y del segundo es 2. Esto es lo que se observa en la parte sombreada de amarillo de la siguiente imagen.

Primero observamos un 2, luego un 1, luego otro 1 y otro 1 y luego un 2 y así sicesivamente. estos son los índices que considera el software para cada categoría. Pero usted se preguntará cómo sé que el 2 corresponde a la categoría femenino y el 1 a masculino. El orden es el mismo que la posición de cada elemento.



Otra forma para saber cual vale 1 y cual vale dos es el orden en el cual aparecen los niveles de la variable. Para saber cuales son los niveles podemos llamar al vector y ver en la última fila.

```
sexo1
```

```
##  [1] MASCULINO FEMENINO  FEMENINO  FEMENINO  MASCULINO FEMENINO  FEMENINO  FEMENINO
##  [9] FEMENINO  FEMENINO  MASCULINO FEMENINO  FEMENINO  MASCULINO MASCULINO MASCULINO
## [17] MASCULINO MASCULINO FEMENINO  MASCULINO FEMENINO  FEMENINO  MASCULINO FEMENINO
## [25] MASCULINO MASCULINO FEMENINO  FEMENINO  FEMENINO  FEMENINO  MASCULINO FEMENINO
## [33] MASCULINO
## Levels: FEMENINO MASCULINO
```

O podemos usar la función `levels()`, para conocer los niveles del vector.[12]

```
levels(sexo1)
```

```
## [1] "FEMENINO"  "MASCULINO"
```

Por las dos formas podemos ver que el primer nivel es `FEMENINO` y el segundo es `MASCULINO`. Es así que el software le asignará el valor de 1 a femenino y 2 a masculino. Aquí surge una pregunta muy interesante ¿Por qué el software R empieza los índices con 1 y no con 0, como lo hacen otros softwares como python, java, stata, entre otros?

La respuesta no es clara al 100%, sólo lo saben los que programaron **R**, pero es una ventaja de R, ya que, si empezaría en 0 el índice al llamar elementos de los vectores. Por ejemplo, si queremos llamar al primer elemento de un vector x usamos `x[1]`, en cambio, en python si queremos llamar al primer elemento tendremos que usar el índice 0, así `x[0]`. A mi parecer, el índice 0 para llamar al primer elemento me resulta extraño y confuso. Para mayor conocimiento y detalle del tema puede ingresar al siguiente enlace stackoverflow

Entonces, la cuestión es ¿se puede modificar el valor de los índices, es decir, que femenino tome 0 y masculino 1, manteniendo el nombre de las categorías? La respuesta es **NO**. Esto no es posible, pero no se observa la utilidad de colocar los índices con 0 y 1. Quizá piense que es necesario cuando desarrollará modelos de regresión o clasificación. En realidad, no, el software R sólo necesita que la variable sea factor o en su defecto que las variables sean convertidas en dummies. Esta cuestión se verá más a detalle en capítulos finales.

Otra cuestión podría ser que a usted le interesa que femenino valga 2 y masculino valga 1. Esto sí es posible, para esto usaremos la función `factor()`.

Primero veamos los argumentos de la función `factor()`.

```
factor(x = character(), levels, labels = levels, exclude = NA, ordered = is.ordered(x),
nmax = NA)
```

Donde:

`x` = Es un vector.
`levels` = Son los niveles del vector x.
`labels` = Son las etiquetas que tomarán los niveles del vector x.
`exclude` = Son los niveles que se excluirán a la hora de convertir en factor al vector x.
`ordered` = Si deseamos que los niveles del vector x tengan un orden.
`nmax` = El número máximo de niveles que se desea tener.

Para verlo a detalle vamos a usar la variable sexo que teníamos creado.

```
sexo
```

```
##  [1] "MASCULINO" "FEMENINO"  "FEMENINO"  "FEMENINO"  "MASCULINO" "FEMENINO"  "FEMENINO"
##  [8] "FEMENINO"  "FEMENINO"  "FEMENINO"  "MASCULINO" "FEMENINO"  "FEMENINO"  "MASCULINO"
## [15] "MASCULINO" "MASCULINO" "MASCULINO" "MASCULINO" "FEMENINO"  "MASCULINO" "FEMENINO"
## [22] "FEMENINO"  "MASCULINO" "FEMENINO"  "MASCULINO" "MASCULINO" "FEMENINO"  "FEMENINO"
```

[12] Si usted no comprende esta sintaxis, le recomendamos revisar el capítulo de vectores aleatorios o buscar cuales son los argumentos de la función `sample()`.

```
## [29] "FEMENINO"  "FEMENINO"  "MASCULINO" "FEMENINO"  "MASCULINO"
```

Por defecto, el software coloca los índices por orden alfabético. Por tal motivo, en los resultados anteriores colocaba a femenino el índice 1 y a masculino el índice 2. Pero si queremos cambiar el orden del indice, es decir, que el índice 2 sea para femenino y el índice 1 para masculino.

Para obtener estos resultados ejecutaríamos la siguiente sintaxis.

```
sexo2<-factor(sexo, levels = c("MASCULINO","FEMENINO"))
```

Lo que hemos realizado es colocar los niveles con el argumento `levels`, es así que como primer nivel hemos considerado a `MASCULINO` y como segundo nivel a `FEMENINO`. Al colocar los niveles en ese orden, el software asignará los indices en ese orden.

```
sexo2
```

```
##  [1] MASCULINO FEMENINO  FEMENINO  FEMENINO  MASCULINO FEMENINO  FEMENINO  FEMENINO
##  [9] FEMENINO  FEMENINO  MASCULINO FEMENINO  FEMENINO  MASCULINO MASCULINO MASCULINO
## [17] MASCULINO MASCULINO FEMENINO  MASCULINO FEMENINO  FEMENINO  MASCULINO FEMENINO
## [25] MASCULINO MASCULINO FEMENINO  FEMENINO  FEMENINO  FEMENINO  MASCULINO FEMENINO
## [33] MASCULINO
## Levels: MASCULINO FEMENINO
```

En efecto, ahora el primer nivel es `MASCULINO` y el segundo es `FEMENINO`. También podemos observar con la función `str()`.

```
str(sexo2)
```

```
##  Factor w/ 2 levels "MASCULINO","FEMENINO": 1 2 2 2 1 2 2 2 2 2 ...
```

Detalle importante a considerar en este punto es que no podemos cometer errores al escribir los niveles, si nosotros escribimos un nivel distinto al que está en nuestra variable, entonces, cometeremos un error y no estaremos considerando a esa categoría en el nivel esperado. Veamos un ejemplo, y observemos lo que sucede.

```
sexo3<-factor(sexo, levels = c("masculino","FEMENINO"))
```

Aquí podemos observar que en vez de colocar el nivel `MASCULINO` se ha colocado en minusculas `masculino`. Lo cual nos genera un error ya que al no encontrar el nivel `masculino` no podrá colocar el índice, y por ende, asiganará el valor de `NA` a cada observación que no tiene el nivel correcto. Observemos.

```
sexo3
```

```
##  [1] <NA>     FEMENINO FEMENINO FEMENINO <NA>     FEMENINO FEMENINO FEMENINO FEMENINO
## [10] FEMENINO <NA>     FEMENINO FEMENINO <NA>     <NA>     <NA>     <NA>     <NA>
## [19] FEMENINO <NA>     FEMENINO FEMENINO <NA>     FEMENINO <NA>     <NA>     FEMENINO
## [28] FEMENINO FEMENINO FEMENINO <NA>     FEMENINO <NA>
## Levels: masculino FEMENINO
```

En efecto, le asignó el valor de `NA` a todos los elementos que no encontró un nivel que se le asigno, en este caso, a todos los que son `MASCULINO`.

Entonces, aquí surge un cuestión ¿podemos cambiar los nombres de los niveles?, la respuesta es sí. Para realizar esto tenemos que usar el argumento `labels`. Veamos un ejemplo, en donde nuestros niveles estén en minúsculas.

```
sexo4<-factor(sexo, levels = c("MASCULINO", "FEMENINO"),
              labels = c("masculino", "femenino"))
```

Hemos hecho el mismo procedimiento que anteriormente, pero ahora estamos utilizando el argumento `labels` que son las etiquetas que se asignarán a cada nivel. Veamos el resultado.

```
sexo4
```

```
##  [1] masculino femenino  femenino  femenino  masculino femenino  femenino  femenino
##  [9] femenino  femenino  masculino femenino  femenino  masculino masculino masculino
## [17] masculino masculino femenino  masculino femenino  femenino  masculino femenino
## [25] masculino masculino femenino  femenino  femenino  femenino  masculino femenino
## [33] masculino
## Levels: masculino femenino
```

Aquí lo fundamental, es el orden. Si nosotros consideramos primero a `MASCULINO` entonces su etiqueta debe de ser `masculino`, si nosotros hubiesemos considerado primero la etiqueta `femenino` en vez de `masculino`, se habría cambiado a todos los `MASCULINOS` por `femenino`. Por tal motivo, es fundamental el orden en el que se coloca los niveles y las etiquetas.

Otra cuestión que puede surgir es el no considerar a uno o más categorías. Para realizar esto vamos a usar el argumento `exclude`. Vamos a crear el vector `x` para desarrollar un nuevo ejemplo.

```
# Creando el vector x.
set.seed(2021)
x<-sample(c("A","B","C","D"), 30, T)
x
```

```
##  [1] "C" "B" "B" "B" "C" "D" "D" "B" "D" "B" "C" "B" "B" "A" "C" "C" "A" "C" "D" "C" "D"
## [22] "B" "C" "D" "A" "C" "B" "D" "B" "D"
```

Hemos creado el vector `x` que tiene 30 elementos, los cuales son las letras: A, B, C y D.

Entonces, si nosotros queremos convertir en factor a la variable `x` y no nos interesa el orden de los índices, entonces podemos correr la siguiente sintaxis.

```
factor(x)
```

```
##  [1] C B B B C D D B D B C B B A C C A C D C D B C D A C B D B D
## Levels: A B C D
```

Al no interesarnos el orden de los índices suficiente con colocar el vector `x` dentro de la función `factor()`. Como sabemos el software, asignará los índices en orden alfabético[13].

El resultado que hemos obtenido es el esperado a asignado los niveles en orden alfabético es así que el primer índice es para `A` y el segundo para `B`, y así sucesivamente.

Pero si nosotros quisíeramos excluir al nivel `B` entonces usaríamos la siguiente sintaxis.

```
factor(x, exclude = "B")
```

```
##  [1] C    <NA> <NA> <NA> C    D    D    <NA> D    <NA> C    <NA> <NA> A    C    C    A
## [18] C    D    C    D    <NA> C    D    A    C    <NA> D    <NA> D
## Levels: A C D
```

Y en efecto, ya no estamos considerando al nivel `B` y asigna el valor de `NA` a todas los elementos que son `B`. Hasta ahora hemos visto ejemplos con factores en donde no es importante el orden de los niveles. Pero en la siguiente sección veremos a detalle como trabajar con niveles ordenados.

6.1.2 Factores con niveles ordenados.

En encuestas se suele preguntas sobre satisfacción, nivel de recomendación o percepción. En este tipo de preguntas el orden de los niveles tiene suma importancia, ya que no es lo mismo sentirse muy satisfecho con sentirse satisfecho. Entonces, si nosotros sabemos que los niveles de una variable se tienen que ordenar

[13] El ciclo while se desarrollará en el siguiente capítulo.

jerarqueicamente, tendremos que convertirlo a una variable factor ordenado. En R para conseguir esto, se puede usar la función `ordered()`.

La función `ordered()` tiene los mismos argumentos que la función `factor()`, así que será sencillo su aplicación. Veamos un ejemplo en donde se tiene la satisfacción sobre el servicio de alcantarillado de un barrio.

```
# Creando la variable que representa la satisfacción
# sobre el servicio de alcantarillado.
satis<-sample(c("MUY SATISFECHO","SATISFECHO","NEUTRAL",
                "INSATISFECHO","MUY SATISFECHO"), 30, T)
satis
```

```
##  [1] "MUY SATISFECHO" "SATISFECHO"     "NEUTRAL"        "INSATISFECHO"   "MUY SATISFECHO"
##  [6] "MUY SATISFECHO" "MUY SATISFECHO" "SATISFECHO"     "NEUTRAL"        "NEUTRAL"
## [11] "SATISFECHO"     "SATISFECHO"     "MUY SATISFECHO" "NEUTRAL"        "SATISFECHO"
## [16] "MUY SATISFECHO" "MUY SATISFECHO" "MUY SATISFECHO" "INSATISFECHO"   "INSATISFECHO"
## [21] "NEUTRAL"        "NEUTRAL"        "SATISFECHO"     "MUY SATISFECHO" "SATISFECHO"
## [26] "MUY SATISFECHO" "MUY SATISFECHO" "MUY SATISFECHO" "MUY SATISFECHO" "MUY SATISFECHO"
```

Hemos creado el vector `satis` que contiene la satisfacción de 30 individuos sobre el servicio de alcantarillado de un barrio. La satisfacción tiene 5 niveles: muy satisfecho, satisfecho, neutral, insatisfecho, muy insatisfecho.

Nosotros tenemos que asignarle valores númericos a cada nivel, ya que en un futuro realizaremos análisis factorial con estas variable y otras. Entonces, si queremos asignarle valores y sabemos que tienen que tener un orden donde muy insatisfecho es el nivel más bajo y muy satisfecho el valor más alto. Tendremos que realizar la siguiente sintaxis.

```
# Convirtiendo en factor ordenado.
satis1<-ordered(satis, levels=c("MUY INSATISFECHO","INSATISFECHO",
                                "NEUTRAL","SATISFECHO","MUY SATISFECHO"))
satis1
```

```
##  [1] MUY SATISFECHO SATISFECHO     NEUTRAL        INSATISFECHO   MUY SATISFECHO
##  [6] MUY SATISFECHO MUY SATISFECHO SATISFECHO     NEUTRAL        NEUTRAL
## [11] SATISFECHO     SATISFECHO     MUY SATISFECHO NEUTRAL        SATISFECHO
## [16] MUY SATISFECHO MUY SATISFECHO MUY SATISFECHO INSATISFECHO   INSATISFECHO
## [21] NEUTRAL        NEUTRAL        SATISFECHO     MUY SATISFECHO SATISFECHO
## [26] MUY SATISFECHO MUY SATISFECHO MUY SATISFECHO MUY SATISFECHO MUY SATISFECHO
## Levels: MUY INSATISFECHO < INSATISFECHO < NEUTRAL < SATISFECHO < MUY SATISFECHO
```

En efecto, hemos obtenido un factor donde los niveles están ordenados de menor a mayor. ¿Cómo se ha hecho? Al usar la función `ordered()` el primer argumento que se tiene que considerar es el vector que queremos ordenar (`satis`) y el segundo argumento son los niveles, pero los niveles tienen que estar en el orden que se desea, es decir, desde el que tendrá el valor más pequeño hasta el que tendrá el valor más grande, por eso los niveles empezó desde muy insatisfecho.

Este resultado también podemos obtenerlo con la función `factor()` siempre y cuando consideremos un argumento adicional, el cual es `ordered = TRUE`. Veamos como se haría lo mismo con la función `factor()`.

```
# Obteniendo el mismo resultado con la función factor().
satis2<-factor(satis, levels = c("MUY INSATISFECHO","INSATISFECHO",
                                 "NEUTRAL","SATISFECHO","MUY SATISFECHO"),
               ordered = T)
satis2
```

```
##  [1] MUY SATISFECHO SATISFECHO     NEUTRAL        INSATISFECHO   MUY SATISFECHO
##  [6] MUY SATISFECHO MUY SATISFECHO SATISFECHO     NEUTRAL        NEUTRAL
## [11] SATISFECHO     SATISFECHO     MUY SATISFECHO NEUTRAL        SATISFECHO
## [16] MUY SATISFECHO MUY SATISFECHO MUY SATISFECHO INSATISFECHO   INSATISFECHO
```

```
## [21] NEUTRAL        NEUTRAL        SATISFECHO     MUY SATISFECHO SATISFECHO
## [26] MUY SATISFECHO MUY SATISFECHO MUY SATISFECHO MUY SATISFECHO MUY SATISFECHO
## Levels: MUY INSATISFECHO < INSATISFECHO < NEUTRAL < SATISFECHO < MUY SATISFECHO
```

Como sabemos al asignarle un orden, los valores tendrán un orden. Lo que nos permitirá hacer futuros cálculos. Es asíq ue si lo queremos ver en números sólo tendríamos que aplicarle la función `as.numeric()` que convierte al factor en un vector numérico.

```
# Convirtiendo a numérico el factor ordenado.
as.numeric(satis1)
```

```
##  [1] 5 4 3 2 5 5 5 4 3 3 4 4 5 3 4 5 5 5 2 2 3 3 4 5 4 5 5 5 5 5
```

En efecto, hemos obtenido los valores numéricos de cada categoría en donde se le asigna el valor de 1 al nivel más pequeño que es `MUY INSATISFECHO` y el valor de 5 al nivel `MUY SATISFECHO`.

6.2 Listas.

Uno de los objetos más importantes en R son las listas, las listas son capaces de almacenar a otros objetos, por ejemplo: vectores, matrices, factores, entre otros. La particularidad de las listas es que no permite que los objetos que contiene pierdan sus características. Es así que si yo tengo una lista de vectores y matrices, los vectores no perderán sus características y seguirán siendo vectores, lo mismo para las matrices.

Veamos un ejemplo en donde vamos a crear una lista que contenga 5 vectores.

```
# Creando los 5 vectores.
x1<-c(1L,2L,5L,7L)
x2<-c("LUZ","OSCURIDAD","NIEBLA")
x3<-c(1.1,4.3,6.6,7.6)
x4<-c(1+4i,6i)
x5<-c(TRUE, FALSE,FALSE, TRUE)
```

Hemos creado 5 vectores: vector de enteros, de caracteres, númerico, complejo y lógico.

Si nosotros queremos tener todo en un mismo objeto, podríamos concatenarlo, pero al hacer eso se perdería la característica de cada vector, ya que lo convertiría en un vector character como se vio en el segundo capítulo.

```
x<-c(x1,x2,x3,x4,x5)

class(x)
```

```
## [1] "character"
```

Entonces, si no queremos que pierda sus características tendremos que integrarlo en una lista. Para crear una lista se usa la función `list()`.

```
# Creando una lista que contenga a los 5 vectores.
lista<-list(x1,x2,x3,x4,x5)

lista
```

```
## [[1]]
## [1] 1 2 5 7
##
## [[2]]
## [1] "LUZ"       "OSCURIDAD" "NIEBLA"
##
## [[3]]
## [1] 1.1 4.3 6.6 7.6
```

```
##
## [[4]]
## [1] 1+4i 0+6i
##
## [[5]]
## [1]  TRUE FALSE FALSE  TRUE
```

La lista que acabamos de crear nos muestra los 5 vectores y cada uno no ha perdido sus características.

6.2.1 ¿Cómo se obtiene cada objeto de una lista?

Una vez que se ha creado la lista usted, puede tener la necesidad de usar uno o varios de los objetos, entonces, usted tiene que llamar a los objetos que contiene sus lista.

El procedimiento es similar al que se usa para llamar elementos de un vector, en esos casos se usaba los corchetes `[]`, para el caso de listas se usa los doble corchetes `[[]]`. Por ejemplo si nosotros queremos llamar al primer vector de nuestra lista, entonces aplicaríamos la siguiente sintaxis.

```
# Llamando al primer objeto de nuestra lista.
lista[[1]]
```

```
## [1] 1 2 5 7
```

en efecto al aplicar doble corchete a la lista y colocar dentro de ella el número 1, estamos llamando al primer objeto.

Como se mencionó anteriormente, al guardarlo en una lista no pierde sus características el vector, por lo cual el primero objeto de la lista es un vector integer.

```
# Viendo si se mantiene las características de los objetos.
class(lista[[1]])
```

```
## [1] "integer"
```

En efecto, es de la clase integer.

Ahora, si queremos llamar al cuarto objeto, entonces, usaríamos la siguiente sintaxis.

```
# Llamando al cuarto objeto de la lista.
lista[[4]]
```

```
## [1] 1+4i 0+6i
```

6.2.2 Asignado nombres a los objetos de una lista.

Los objetos de una lista admiten nombres, por lo que se les puede asignar nombres y luego llamarlos por esos nombres. Con la función `names()` se obtiene o se les asigna nombres. Primero apliquemos la función `names()` a la lista que hemos creado con el fin de obtener los nombres de la lista, como sabemos que no tienen nombres nuestros objetos nos arrojará el valor de NULL.

```
# Obteniendo los nombres de nuestra lista.
names(lista)
```

```
## NULL
```

En efecto, no arroja NULL. Lo que vamos hacer ahora es asignarle nombres. Los nombres serán las clases de cada vector.

```
# Asignando nombres a nuestra lista.
names(lista)<-c("entero","caracter","numerico","complejo","logico")
```

Veamos si nuestra lista ya tiene nombres.

```
names(lista)
```

```
## [1] "entero"   "caracter" "numerico" "complejo" "logico"
```

En efecto, ya tienen nombres, por lo cual ya podremos llamarlos por sus nombres. Por ejemplo, vamos a llamar al objeto llamado `caracter`, pero para llamarlo por sus nombres, ya no se necesitará usar el doble corchete `[[]]`, sino el símbolo de dólar `$`.

```
# Llamando al objeto llamado caracter.
lista$caracter
```

```
## [1] "LUZ"       "OSCURIDAD" "NIEBLA"
```

El resultado, es el objeto que contiene elementos caracter. Entonces podemos llamar a los objetos de una lista de dos formas con el doble corchete `[[]]` cuando tenga o no tenga nombres nuestros objetos o con el `$` siempre y cuando los objetos tengan nombres.

Como último ejemplo, veremos como llamar al segundo elemento del vector llamado entero de nuestra lista.

```
# Llamando al segundo elemento del vector entero de la lista.
lista$entero[2]
```

```
## [1] 2
```

```
# O también podemos llamarlo con:
lista[[1]][2]
```

```
## [1] 2
```

Al final obtenemos el mismo resultado.

En el siguiente capítulo se verá el tema de matrices, desde como crearlas hasta llamar elementos.

7 MATRICES.

El uso de matrices es muy importante para resolver problemas de optimización estática y dinámica. Pero si se tiene un perfil estadístico su uso es menor, ya que la mayoría de procedimientos estadísticos ya están desarrollados en `packages`. El principal uso que se le da es para la creación de funciones, para aprender como se selecciona filas y columnas, esto nos dará una visión muy amplia cuando tengamos que crear un data frame, que se verá en los siguientes capítulos.

7.1 Creación de matrices.

Al igual que un curso de álgebra lineal en el software R la construcción de matrices se hace a través de vectores. La función que crea matrices es la función `matrix()`, que tiene los siguientes argumentos:

```
matrix(data = NA, nrow = 1, ncol = 1, byrow = FALSE, dimnames = NULL)
```

Donde:
`data` = Es un vector.
`nrow` = El número de filas que tendrá la matriz, el valor de defecto es de 1 fila.
`ncol` = El número de columnas que tendrá la matriz, el valor de defecto es de 1 columna.
`byrow` = Es un argumento lógico que si toma el valor de TRUE, entonces, creará la matriz ordenando los elementos por filas. Su valor por defecto es FALSE, lo que implica que crea la matriz ordenando los elementos por columnas.
`dimnames` = Es una lista en donde se colocará los nombres de las filas y columnas. Su valor por defecto es NULL, lo que implica que las filas como las columnas no tendrán nombres.

Veamos un sencillo ejemplo en donde crearemos una matriz de 2x2.

```
# Creando una matriz 2x2.
matrix(1:4, nrow = 2)
```

```
##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4
```

En efecto, se ha creado la matriz de 2x2 en donde se usó los elementos del 1 al 4 (`1:4`) y se indicó que el número de filas sea de 2 (`nrow = 2`). Además ordenó los elementos por columnas, ya que el valor por defecto de byrow es FALSE (`byrow = FALSE`).

Ahora veamos como crear una matriz 3x2, es decir, que tenga 3 filas y 2 columnas.

```
# Creando la matriz 3x2.
matrix(1:6, nrow = 3, ncol = 2)
```

```
##      [,1] [,2]
## [1,]    1    4
## [2,]    2    5
## [3,]    3    6
```

Se ha creado la matriz 3x2, en este caso se ha tenido que especificar el número de filas con el argumento `nrow = 3` y el número de columnas con el argumento `ncol = 2`.

El siguiente ejemplo, será similar al anterior pero en este caso vamos a usar el argumento `byrow`. Lo que implica que se ordenará los elementos por filas.

```
# Creando una matriz donde los elementos se ordenan por filas.
matrix(1:6, nrow = 3, ncol = 2, byrow = T)
```

```
##      [,1] [,2]
## [1,]    1    2
## [2,]    3    4
## [3,]    5    6
```

En efecto, ahora los elementos se han completado por filas, anteriormente este ordenamiento se daba por columnas.

Por último, veremos un ejemplo en donde las filas como las columnas tengan nombres.

```
# Creando una matriz donde las filas y columnas tengan nombres.
matrix(1:6, nrow = 3, ncol = 2, byrow = T,
       dimnames = list(c("Año 2020","Año 2021","Año 2022"),
                       c("Sexo Masculino","Sexo Femenino")))
```

```
##          Sexo Masculino Sexo Femenino
## Año 2020              1             2
## Año 2021              3             4
## Año 2022              5             6
```

Ahora, veamos el caso en donde se desea crear una matriz 4x4 pero sólo se le brinda un vector de 10 elementos. En este caso el número de elementos es inferior al que requerirá una matriz 4x4, lo que hará el software es completar los elementos faltantes con los primeros elementos del vector hasta completar 16 elementos.

```
# Creando una matriz con menos elementos que la matriz.
matrix(1:10, ncol = 4, nrow = 4)
```

```
## Warning in matrix(1:10, ncol = 4, nrow = 4): la longitud de los datos [10] no es un
## submúltiplo o múltiplo del número de filas [4] en la matriz
```

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

Si nos fijamos bien, agregó los elementos 1, 2, 3, 4, 5, 6 en la matriz, ya que sólo se le brindó 10 elementos. Asimismo, nos ha arrojado un mensaje de advertencia en donde nos dice que `la longitud de los datos [10] no es un submúltiplo o múltiplo del número de filas [4] en la matriz`.

Resultado contrario arrojará cuando se le asigne un vector con más elementos que elementos de la matriz. Por ejemplo se le brindará 10 elementos, pero le especificamos una matriz de 2x2.

```
# Creando una matriz con más elementos que la matriz.
matrix(1:10, ncol = 2, nrow = 2)
```

```
##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4
```

Sólo a colocado los 4 primeros elementos.

7.2 Concatenar matrices por filas o columnas.

Si nosotros queremos juntar matrices por filas, es decir, apilar matrices por filas entonces usaremos la función `rbind()`. Veamos una ejemplo.

Vamos a juntar 2 matrices por las filas. Para lo cual creamos las dos matrices.

```
# La primera matriz.
m1<-matrix(c("Azul","Rojo","Amarillo","Verde"), ncol = 2)
m1
```

```
##      [,1]   [,2]
## [1,] "Azul" "Amarillo"
## [2,] "Rojo" "Verde"
```

```
# La segunda matriz.
m2<-matrix(1:4, ncol = 2)
m2
```

```
##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4
```

La primera matriz se guardó en el objeto `m1` y la segundo en el objeto `m2`. Las vamos a concatenar por filas, es decir, vamos a crear una matriz 4x2.

```
# concatenando las matrices por filas.
rbind(m1,m2)
```

```
##      [,1]   [,2]
## [1,] "Azul" "Amarillo"
## [2,] "Rojo" "Verde"
## [3,] "1"    "3"
## [4,] "2"    "4"
```

En efecto, se tiene un matriz 4x2.

Ahora veamos el caso de concatenar matrices por columnas, para esto se usará la función `cbind()`. Vamos a concatenar 2 matrices 3x2 para que el resultado sea una matriz 3x4.

Primero vamos a crear las 2 matrices 3x2.

```
# Primera matriz.
ma1<-matrix(c("A","B","C","D","E","F"), ncol = 2, nrow = 3)
ma1
```

```
##      [,1] [,2]
## [1,] "A"  "D"
## [2,] "B"  "E"
## [3,] "C"  "F"
```

```
# Segunda matriz.
ma2<-matrix(2012:2017, ncol = 2, nrow = 3)
ma2
```

```
##      [,1] [,2]
## [1,] 2012 2015
## [2,] 2013 2016
## [3,] 2014 2017
```

Hemos creado las 2 matrices 3x2 que han sido guardadas en los objetos `ma1` y `ma2`. Ahora los concatenamos por columnas con la función `cbind()`.

```
# Concatenando las matrices por columnas.
cbind(ma1,ma2)
```

```
##      [,1] [,2] [,3]   [,4]
## [1,] "A"  "D"  "2012" "2015"
## [2,] "B"  "E"  "2013" "2016"
## [3,] "C"  "F"  "2014" "2017"
```

En efecto, se ha creado la matriz 3x4.

El uso de las funciones `rbind()` y `cbind()` también se podrá usar cuando queramos juntar data frames.

7.3 Selecionar elementos de una matriz.

En capítulos anteriores se seleccionaba elementos de vectores usando corchetes `[]` y de las listas usando doble corchete `[[]]` o con el símbolo de dólar `$`. En el caso de matrices también se usan los corchetes `[]`, pero dentro de ellos debe de ir un indicador de la fila y la columna del vector. La estructura será la siguiente:

`matriz[fila,columna]`

Si queremos seleccionar elementos de la matriz, primero se colocará el objeto matriz, luego los corchetes, dentro de estos corchetes dos números: el primero indica la fila y el segundo la columna de la matriz.

Para ver ejemplos, crearemos una matriz 5x4 con elementos aleatorios.

```
# Creando la matriz 5x4.
set.seed(2021)
m<-matrix(sample(1:100, 20), ncol = 4, nrow = 5)
m
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    7   70   76    3
## [2,]   38   64   51   98
## [3,]   46   99   60   26
```

```
## [4,]   58   69   18   36
## [5,]   12   23   88   31
```

La matriz que se ha creado se ha guardado en el objeto `m`.

Si queremos seleccionar el elemento de la fila 1 y columna 2, la sintaxis sería la siguiente:

```
# Seleccionando el elemento de la fila 1 y columna 2.
m[1,2]
```

```
## [1] 70
```

con `m` llamamos a la matriz, con `[1,2]` llamamos al elemento de la primera fila y de la segunda columna. El resultado es 70.

A continuación se muestra dos ejemplos adicionales de selección de elementos.

```
# Seleccionando el elemento de la fila 3 y columna 4.
m[3,4]
```

```
## [1] 26
```

```
# Seleccoando el elemento de la fila 4 y columna 2.
m[4,2]
```

```
## [1] 69
```

Si queremos seleccionar todo una fila o toda una columna.

```
# Seleccionado toda la columna 2.
m[,2]
```

```
## [1] 70 64 99 69 23
```

```
# Seleccioando toda la fila 3.
m[3,]
```

```
## [1] 46 99 60 26
```

Si se quiere seleccionar más de una fila o columna.

```
# Seleccioando las filas 2 y 3.
m[2:3,]
```

```
##      [,1] [,2] [,3] [,4]
## [1,]   38   64   51   98
## [2,]   46   99   60   26
```

```
# Seleccioando las columnas 1 y 4.
m[,c(1,4)]
```

```
##      [,1] [,2]
## [1,]    7    3
## [2,]   38   98
## [3,]   46   26
## [4,]   58   36
## [5,]   12   31
```

Si se quiere excluir filas o columnas.

```
# Excluyendo la fila 2.
m[-2,]
```

```
##      [,1] [,2] [,3] [,4]
```

```
## [1,]    7   70   76    3
## [2,]   46   99   60   26
## [3,]   58   69   18   36
## [4,]   12   23   88   31
```

```
# Excluyendo la columna 1.
m[,-1]
```

```
##      [,1] [,2] [,3]
## [1,]   70   76    3
## [2,]   64   51   98
## [3,]   99   60   26
## [4,]   69   18   36
## [5,]   23   88   31
```

```
# Excluyendo las columnas 2 y 4.
m[,-c(2,4)]
```

```
##      [,1] [,2]
## [1,]    7   76
## [2,]   38   51
## [3,]   46   60
## [4,]   58   18
## [5,]   12   88
```

La forma en la que se ha seleccionado elementos, filas o columnas será el mismo que usaremos para hacer selecciones en los objetos data frame.

7.4 Dimensiones de la matriz.

Las matrices tiene filas y columnas. Si nosotros nos encontramos con un objeto que es una matriz, sabremos el número de filas y el de columnas con la función `dim()`.

Por ejemplo deseamos saber cual es el número de filas y columnas de la matriz `m`.

```
# Para saber el número de filas y columnas de la matriz.
dim(m)
```

```
## [1] 5 4
```

Nos indica que son 5 filas y 4 columnas. Ahora si queremos saber sólo el número de filas, entonces usaremos la siguiente sintaxis.

```
# Para saber el número de filas.
dim(m)[1]
```

```
## [1] 5
```

Y para saber el número de columnas.

```
# Para saber el número de columnas.
dim(m)[2]
```

```
## [1] 4
```

Y para saber el número de elementos de la matriz usaremos la función `length()`

```
# Para saber el número de elementos de la matriz.
length(m)
```

```
## [1] 20
```

Por último, para saber si un objeto es una matriz. Usaremos la función `is.matrix()`.

```
is.matrix(m)
```

```
## [1] TRUE
```

En el siguiente capítulo se mostrará como realizar operaciones matemáticas y estadísticas con matrices.

8 ÁLGEBRA MATRICIAL.

En este capítulo se desarrollará operaciones matemáticas con matrices. Y en la parte final se muestra como resolver un sistema de ecuaciones lineales.

8.1 Operaciones básicas.

En la siguiente tabla se muestran las operaciones y su sintaxis.

OPERACIÓN	SINTAXIS
Adición	+
Sustracción	-
Multiplicación por un escalar	*
Producto de Matrices	%*%
Potencia de una matriz	mtx.exp()
Producto exterior	%o%
Producto Kronecker	%x%

Primero, crearemos un par de matrices que nos permitirán desarrollar estas operaciones.

```
# Primera matriz.
A<-matrix(1:4, ncol = 2)
A
```

```
##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4
```

```
# Segunda matriz.
B<-matrix(c(4,2,0,1), ncol = 2)
B
```

```
##      [,1] [,2]
## [1,]    4    0
## [2,]    2    1
```

Si se desea sumar matrices entonces se usa el operador `+`.

```
# Sumando las 2 matrices.
A+B
```

```
##      [,1] [,2]
## [1,]    5    3
## [2,]    4    5
```

si usted desea restar las matrices, tendrá que usar el operador `-`.

```
# Restando las 2 matrices.
A-B
```

```
##      [,1] [,2]
## [1,]   -3    3
## [2,]    0    3
```

Si se desea multiplicar por un escalar a una matriz, entonces se usará el operador `*`. Por ejemplo, vamos a multiplicar 5 por la matriz A.

```
# Multiplicación de un escalar por una matriz.
5*A
```

```
##      [,1] [,2]
## [1,]    5   15
## [2,]   10   20
```

Para el caso de multiplicación de matrices, se tendrá que usar el operador `%*%`.

```
# Multiplicación de matrices.
A%*%B
```

```
##      [,1] [,2]
## [1,]   10    3
## [2,]   16    4
```

También podríamos multiplicar la matriz A tres veces.

```
# Multiplicación de A por A por A.
A%*%A%*%A
```

```
##      [,1] [,2]
## [1,]   37   81
## [2,]   54  118
```

La lógica indica que si deseamos multiplicar la matriz A 4 veces, tendremos que usar `A%*%A%*%A%*%A`. Y si deseamos multiplicar la matriz A 5 veces `A%*%A%*%A%*%A%*%A`. Pero esto puede resultar tedioso si desea usted desea elevar la matriz A a la 10 o a la 15 o a un exponente mayor. Entonces, se tiene que crear una función que permita desarrollar este cálculo.

Planteo la siguiente función para la solución de este tipo de problemas.

```
matriz_n<-function(x, n){
  y<-1
  t<-x
  while (y+1<=n) {
    t<-t%*%x
    y<-y+1
  }
  t
}
```

Para la construcción de esta función estoy usando funciones como `while()` que se verá en el siguiente capítulo, asimismo, en los capítulos finales se mostrará como crear funciones para solucionar cualquier tipo de problema.

Pongamos en práctica la función que se acaba de crear. Vamos a elevar la matriz A al cubo.

```
# La matriz A al cubo.
matriz_n(A, 3)
```

```
##      [,1] [,2]
```

```
## [1,]   37   81
## [2,]   54  118
```

El resultado es igual al que se haría manualmente.

```
A%*%A%*%A
```

```
##      [,1] [,2]
## [1,]   37   81
## [2,]   54  118
```

Esto es una muestra de cómo podríamos solucionar este problema creando nuestras propias funciones, lo que es el reflejo fiel de la palabra programar.

Pero el problema es sencillo y otros usuarios de R, ya han programado una función similar a la que hemos desarrollado y lo tienen documentado en un paquete.

Un paquete en R, es una compilación de funciones que han sido creadas por los usuarios y se pueden descargar desde el CRAM de R. Para instalar un paquete tendremos que usar la función `install.package()`.

Como ejemplo, vamos a instalar el paquete `Biodem` que tiene una función que calcula la potencia de las matrices.

```
# Instalar el paquete Biodem.
install.packages("Biodem")
```

Los paquetes sólo se instalan una vez y nada más. El siguiente paso es cargar el paquete en nuestro entorno de R. Para esto se usa la función `library()`.

```
library(Biodem)
```

Se tendrá que cargar los paquetes cada vez que inicies sesión en R, es decir, cada vez que usted abra el software.

Una vez cargado el paquete llamaremos a la función `mtx.exp()` que nos permite calcular las potencias de las matrices. Vamos a calcular lo mismo que hicimos con la función que hemos creado, es decir, elevar la matriz A al cubo.

```
# Elevar la matriz al cubo.
mtx.exp(A,3)
```

```
##      [,1] [,2]
## [1,]   37   81
## [2,]   54  118
```

Entonces, hay dos posibilidades. La primera es crear nuestra propia función o descargar un paquete en donde haya una función que permita realizar el cálculo que deseamos

Un punto interesante es el tiempo que realiza nuestro procesador para realizar el cálculo, pero esto lo veremos en los anexos.

Por otro lado, si nosotros deseamos calcular el producto exterior de las matrices.

```
# Producto exterior de la matriz A.
A%o%A
```

```
## , , 1, 1
##
##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4
##
## , , 2, 1
```

```
##
##      [,1] [,2]
## [1,]    2    6
## [2,]    4    8
##
## , , 1, 2
##
##      [,1] [,2]
## [1,]    3    9
## [2,]    6   12
##
## , , 2, 2
##
##      [,1] [,2]
## [1,]    4   12
## [2,]    8   16
```

El producto exterior de matrices nos servirá para el cálculo del producto Kronecker. En ese sentido, para hallar el producto Kronecker se tendrá que usar el operador `%x%`.

```
# El producto Kronecker de la matriz A con ella misma.
A%x%A
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    3    3    9
## [2,]    2    4    6   12
## [3,]    2    6    4   12
## [4,]    4    8    8   16
```

8.2 Principales operaciones con matrices.

En la siguiente tabla se muestran las principales operaciones de las matrices y su sintaxis en el software R.

OPERACIÓN	SINTAXIS
Transpuesta	**t()**
Diagonal	**diag()**
Traza	**sum(diag())**
Determinante	**det()**
Inversa	**solve()**
Descomposición	**qr()**
Rango	**qr()$rank**
Descomposición de cholesky	**chol()**
Varianza	**var()**

Si nosotros queremos calcular la transpuesta de la matriz A, entonces, usaremos la función `t()`.

```
# La matriz A.
A
```

```
##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4
```

```r
# Su transpuesta.
t(A)
```

```
##      [,1] [,2]
## [1,]    1    2
## [2,]    3    4
```

Asimismo, si se quere calcular la diagonal de la matriz A, tendremos que usar la función `diag()`.

```r
# La diagonal de la matriz A.
diag(A)
```

```
## [1] 1 4
```

Como sabemos la traza de una matriz es la suma de la diagonal. Entonces, si queremos calcular la traza de la matriz, sólo tendríamos que sumar los valores de los elementos de la diagonal.

```r
# Calculando la traza de la matriz.
sum(diag(A))
```

```
## [1] 5
```

Pero si usted desea crear la función para la traza, lo podría hacer de la siguiente manera.

```r
# Creando la función que calcule la traza.
traza<-function(x){
  sum(diag(x))
}

# Aplicando la función que calcula la traza.
traza(A)
```

```
## [1] 5
```

Lo que nos resulta en el mismo resultado.

Uno de los calculos muy importantes en matrices cuadradas es el cálculo de la determinante. En R para calcular la determinante con la función `det()`.

```r
# Calculando la determinante de la matriz A.
det(A)
```

```
## [1] -2
```

Asimismo, si deseamos calcular la inversa de la matriz, entonces, usaremos la función `solve()`.

```r
# Calculando la inversa de la matriz A.
solve(A)
```

```
##      [,1] [,2]
## [1,]   -2  1.5
## [2,]    1 -0.5
```

En álgebra matricial también es muy importante descomponer las matrices (factorizar), en R se puede descomponer matrices con el método de descomposicón QR el cual es el producto de la matriz Q (matriz ortogonal) y la matriz R (matriz triángulo superior). Para lo cual se usa la función `qr()`.

```r
# Calculando la descomposición de una matriz.
qr(A)
```

```
## $qr
##            [,1]       [,2]
## [1,] -2.2360680 -4.9193496
```

```
## [2,]  0.8944272 -0.8944272
##
## $rank
## [1] 2
##
## $qraux
## [1] 1.4472136 0.8944272
##
## $pivot
## [1] 1 2
##
## attr(,"class")
## [1] "qr"
```

Primero si nos damos cuenta el resultado es una lista que contiene 4 objetos.

```
class(qr(A))
```

```
## [1] "qr"
```

Al ser una lista podemos llamar a cada uno de los objetos con el doble corchete `[[]]` o con el dólar `$` usando el nombre de los objetos.

El objeto `qr` es una matriz que contiene a la matriz R, es el triángulo superior de esta matriz y a la matriz Q, que es el triángulo superior pero de forma compacta.

Si nosotros queremos ver cuales son las matrices Q y R específicamente, tendremos que usar las funciones `qr.Q()` y `qr.R()`, en donde el argumento de las dos funciones es el objeto `qr`.

```
# Calculando la matriz Q.
qr.Q(qr(A))
```

```
##            [,1]       [,2]
## [1,] -0.4472136 -0.8944272
## [2,] -0.8944272  0.4472136
```

```
# Calculando la matriz R.
qr.R(qr(A))
```

```
##           [,1]       [,2]
## [1,] -2.236068 -4.9193496
## [2,]  0.000000 -0.8944272
```

Perfecto, hemos obtenido la matriz Q que es una matriz ortogonal y la matriz R que es una matriz triángulo superior.

Asimismo, podemos hacer la comprobación. Sabemos que ha descompuesto nuestra matriz A en dos matrices en: Q y R. Entonces si multplicamos la matriz Q por la matriz R deberíamos de obtener la matriz A.

```
# Comprobando que la descomposición es la correcta.
qr.Q(qr(A))%*%qr.R(qr(A))
```

```
##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4
```

En efecto, tenemos la matriz A, por lo cual el método de descomposición queda comprobado.

Ahora si deseamos saber el rango de la matriz, usaremos la misma función `qr()` y llamaremos al objeto `rank`.

```r
# Calculando el rango de la matriz.
qr(A)$rank
```

```
## [1] 2
```

Que nos indica que es de rango 2.

También podemos descomponer las matrices por el método de cholesky, pero lo primordial para poder desarrollar este método es que la matriz se simétrica y definida positiva. Por lo que para mostrar un ejemplo, vamos a crear una matriz simétrica y definida positiva.

```r
C<-matrix(c(4,1,1,4), ncol = 2)
C
```

```
##      [,1] [,2]
## [1,]    4    1
## [2,]    1    4
```

El método de Cholesky dice que descompondrá a la matriz en dos matrices, la primera en una matriz triángulo inferior y la segunda es la transpuesta de la matriz triángulo inferior.

$$X = R * R'$$

En el software R se usa la función `chol()`, pero el resultado que arroja es la matriz triángulo superior, ya que usa la siguiente fórmula para descomponer.

$$X = R' * R$$

Donde: R es la matriz triángulo inferior.

Es así que el software R arroja R', por eso se convierte en una matriz triángulo superior.

```r
# Determinando la matriz triángulo superior por la descompisición
# de Cholesky.
chol(C)
```

```
##      [,1]     [,2]
## [1,]    2 0.500000
## [2,]    0 1.936492
```

Comprobando que el resultado de la factorización es el correcto.

```r
# Comprobando.
t(chol(C))%*%chol(C)
```

```
##      [,1] [,2]
## [1,]    4    1
## [2,]    1    4
```

Y en efecto, obtenemos la matriz original, por lo que la descomposición de Cholesky es la correcta.

Por último, para el cálculo de la matriz de varianzas y covarianzas se usará la función `var()`. Vamos a calcular la matriz de varianas y covarianzas de la matriz A.

```r
var(A)
```

```
##      [,1] [,2]
## [1,]  0.5  0.5
## [2,]  0.5  0.5
```

8.3 Solución a sistemas de ecuación.

El software R también nos permite resolver sistemas de ecuaciones. Por ejemplo, si tenemos el siguiente sistema de ecuaciones.

$$2x + 3y = 1$$

$$3x - 7y = 2$$

Para poder desarrollarlo se puede hacer mediante el cálculo matricial, lo que se tendrá que hacer es convertir las dos ecuaciones en matrices.

$$\begin{pmatrix} 2 & 3 \\ 3 & 7 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \end{pmatrix}$$

Las matrices que nos interesan son la primera y la tercera. Las creamos.

```
# La primera matriz.
D<-matrix(c(2,3,3,-7), nrow = 2, byrow = TRUE)
D
```

```
##      [,1] [,2]
## [1,]    2    3
## [2,]    3   -7
```

```
# La segunda Matriz.
v<-matrix(1:2, ncol = 1)
v
```

```
##      [,1]
## [1,]    1
## [2,]    2
```

Como se puede ver las matrices son iguales a las ecuaciones que se muestran.

Para poder obtener respuesta se hace usa la función `solve()`.

```
# Para solucionar el sistema de ecuaciones.
solucion<-solve(D,v)
solucion
```

```
##             [,1]
## [1,]  0.56521739
## [2,] -0.04347826
```

Guardamos el resultado en el objeto `solucion` y podemos ver que nos arroja los valores de `x` y `y` que resuelven el sistema de ecuaciones.

Podemos cambiar los nombres de las filas con el fin de obtener un resultado más accesible a los ojos.

```
dimnames(solucion)<-list(c("x", "y"), NULL)
solucion
```

```
##          [,1]
## x  0.56521739
## y -0.04347826
```

Perfecto, ahora se muestran los resultados de `x` y `y` que resuelven las ecuaciones.

8.4 Valores y vectores propios.

Si usted desea calcular, los valores y vectores propios de una matriz tendrá que usar la función `eigen()`.

Para el ejemplo se calculará los valores y vectores propios de la matriz `C`, la que se usó para la factorización con el método de cholesky.

```
# Para el cálculo de valores y vectores propios.
eigen(C)
```

```
## eigen() decomposition
## $values
## [1] 5 3
##
## $vectors
##           [,1]       [,2]
## [1,] 0.7071068 -0.7071068
## [2,] 0.7071068  0.7071068
```

El objeto que hemos obtenido es una lista, por lo cual, si sólo deseamos los valores propios se tendría que usar la siguiente sintaxis `eigen()$values`. Asimismo, si sólo deseo los vectores propios, la sintaxis que se tendría que usar sería`eigen()$vectors`.

```
# Sólo para los valores propios.
eigen(C)$values
```

```
## [1] 5 3
```

```
# Sólo para los vectores propios.
eigen(C)$vectors
```

```
##           [,1]       [,2]
## [1,] 0.7071068 -0.7071068
## [2,] 0.7071068  0.7071068
```

En los últimos capítulos se verá aplicaciones en economía, específicamente en la maximización de la utilidad del consumidor, minimización de los costos de producción, modelos de regresión lineal, entre otras, usando matrices. Por el momento no podemos hacerlo, ya que aún no hemos desarrollado el capítulo de calculo diferencial.

9 CONDICIONAL IF ELSE.

En este capítulo se muestra la condicional `if` que ejecuta un proceso siempre y cuando la condición lógica es verdadera. Esta condicional es fundamental en cualquier lenguaje de programación. Asimismo, si se complementa con la condicional `else` o `else if` se aborda la totalidad de los casos a la hora de programar un proceso.

9.1 Condicional if.

La condicional `if` sirve para ejecutar procesos cuando se cumple una condición lógica que se le asigna. En R tiene la siguiente sintaxis.

```
if("condición lógica"){"Proceso a ejecutarse"}
```

Siempre y cuando la condición lógica se cumpla se ejecutará el proceso especificado.

Veamos un ejemplo en donde usamos la condicional `if`. Para lo cual vamos a definir un vector.

```r
# Definiendo el vector x.
x<-10
x
```

```
## [1] 10
```

Hemos creado el vector `x` que contiene al elemento 10.

Vamos a consultarle al programa si el vector x es igual a 10 y si es correcto que imprima la oración "El vector x vale 10".

```r
# ¿x es igual a 10?
if(x==10){
  print("El vector x vale 10")
}
```

```
## [1] "El vector x vale 10"
```

En efecto, el software a impreso la oración "El vector x vale 10". Pero veamos como se obtuvo este resultado. Primero, se colocó la condición lógica `x==10` y al ser `TRUE` entonces procedió a ejecutar el proceso el cual era `print("El vector x vale 10")`. La función `print()` se usa para imprimir el contenido de los objetos.

Ahora veamos que sucede si la condicional no es `TRUE` sino `FALSE`. Para esto vamos a consultarle al software si x es igual a 100 y si es correcto que imprima la oración "El vector x vale 100".

```r
# ¿x es igual a 100?
if(x==100){
  print("El vector x vale 100")
}
```

No nos arroja absolutamente nada. Esto sucede porque no se cumple la condición lógica lo que implica que no se ejecutará el proceso que se encuentra entre llaves `print("El vector x vale 100")`.

Ahora hagamos una aplicación a un sorteo. Vamos a escoger a un individuo de un total de 5 personas y si el individuo ganador es "Luis", entonces le diremos al software que imprima la oración "El ganador del sorteo es Luis".

```r
# Creando el vector de personas que participarán en el sorteo.
x<-c("Juan","David","Luis","Marcos","Manuel")
```

El siguiente paso es escoger a un ganador, para esto se usará la función `sample()`.

```r
# Escogiendo al ganador.
set.seed(2021)
y<-sample(x, 1, F)
```

Por último le consultaremos al software si "Luis" es el ganador.

```r
if(y=="Luis"){
  print("El ganador del sorteo es Luis")
}
```

No nos imprime nada, lo que implica que el ganador no era Luis. Veamos cual fue la persona que ganó el sorteo.

```r
# La persona que ganó el sorteo.
y
```

```
## [1] "David"
```

El ganador fue David. Lo que generó que no imprima el nombre de Luis como ganador del sorteo.

Ahora veamos otro ejemplo en donde vamos a crear una función que imprima nombres siempre y cuando la edad de la persona es menor a 30 años. Para lo cual vamos a crear una función.

```
# Creando la función que imprime nombres.
imp_nombre<-function(nombre,edad){
  if(edad<30){
    print(paste0("El nombre es: ",nombre))
  }
}
```

Nuestra función `imp_nombre` tiene dos argumentos: nombre y edad. Lo que hace nuestra función es que si la edad es menor que 30, entonces imprimirá `El nombre es .....`. Si no es así, entonces, no imprimirá nada porque no se cumple la condicional.

Veamos un ejemplo en donde la edad es 17 y el nombre es Natalia.

```
# ¿Imprimirá el nombre de Natalia, si ella tiene 17 años?
imp_nombre(nombre = "Natalia", edad = 17)
```

```
## [1] "El nombre es: Natalia"
```

Imprimió el nombre ya que 17 es menor que 30.

9.2 Condicional if else.

La condicional `if else` sirve para ejecutar procesos donde haya dos escenarios. Es decir, si se cumple la condición el software tiene que ejecitar el proceso, pero si no se cumple, entonces el software tiene que ejecutar otro proceso. Para entender mejor, veamos primero como es la sintaxis de esta condicional.

```
if("Condición lógica"){"Se realiza el proceso si cumple la condición"} else {"Se realiza el proceso si no se cumple la condición" }
```

Retomemos el ejemplo del sorteo. En el cual creamos un vector de nombres y se realizó el sorteo, en donde si el ganador era Luis, entonces, imprimiría la oración `El ganador del sorteo es Luis`. Ahora lo que haremos es agregar la condicional `else` y que imprima el nombre del ganador si el ganador no es Luis.

Primero volveremos a correr el código del sorteo.

```
# Escogiendo al ganador.
set.seed(2021)
y<-sample(x, 1, F)
```

El ganador se encuentra en el vector `y`.

Ahora agregando la condicional `else`.

```
# En el ejemplo del sorteo, agregando la condicional else.
if(y=="Luis"){
  print("El ganador del sorteo es Luis")
} else {
  print(paste0("El ganador del sorteo es ", y))
}
```

```
## [1] "El ganador del sorteo es David"
```

Nos arroja que el ganador es David. Esto se da porque la condicional no se `TRUE`, entonces, tiene que ejecutar el proceso que se encuentra en la condicional `else`.

Veamos el mismo ejemplo, pero ahora cambiemos la semilla para obtener otro resultado. Probemos con la semilla `1010`.

```r
# Cambiando la semilla para obtener otro resultado.
# Escogiendo al ganador.
set.seed(1010)
y<-sample(x, 1, F)

# Encontrando al ganador.
if(y=="Luis"){
  print("El ganador del sorteo es Luis")
} else {
  print(paste0("El ganador del sorteo es ", y))
}
```

```
## [1] "El ganador del sorteo es Marcos"
```

Nos arroja que el ganador del sorteo es Marcos. Ya que otra vez, la condicional no es `TRUE`.

Veamos un ejemplo adicional en donde agregemos la condicional `else` al ejemplo de imprimir el nombre. En este ejemplo siempre y cuando la edad era menor a 30 se imprimía el nombre. En este caso tendrá que imprimir la oración `La edad es mayor a 30. No se imprimirá el nombre`, siempre y cuando no se cumpla la condición de que la edad sea menor que 30.

```r
# Agregando la condicional else a la función.
imp_nombre<-function(nombre,edad){
  if(edad<30){
    print(paste0("El nombre es: ",nombre))
  } else {
    print("La edad es mayor a 30. No se imprimirá el nombre")
  }
}
```

Ahora probemos con Ítalo que tiene 52 años ¿Cuál será el resultado?

```r
# Probemos si imprime el nombre de Ítalo que tiene 52 años.
imp_nombre(nombre = "Ítalo", edad = 52)
```

```
## [1] "La edad es mayor a 30. No se imprimirá el nombre"
```

Al ser la edad de Ítalo 52 no se cumple la condición porque tendrá que ejecutar el proceso que se encuntre en `else`. Por eso es que nos arroja `"La edad es mayor a 30. No se imprimirá el nombre"`.

Hasta el momento hemos visto sólo como cubrir dos escenarios, el que resultaría si se cumple la condición y el que pasaría si no se cumple. Pero en la vida real, no sólo se tiene un par de escenarios sino muchos más. Para abordar estos múltiples escenarios usaremos `if else` anidados.

9.3 Condicional if else anidado.

Como se mencionó cuando se quiere cubrir más de 2 escenarios se tendrá que recurrir a `if else` anidados. La sintaxis para tres escenarios es la siguiente.

```
# if("Condición 1"){"Se realiza el proceso si cumple la condición 1"} else if ("Condición
2") {" Se realiza el proceso si no se cumple la condición 1 y si la condición 2"}  else
{"Se realiza el proceso si no se cumple la condición 1 y la condición 2"}
```

Pero esto se puede aplicar para más escenarios.

Primero veamos un ejemplo. Retomemos el ejemplo de la impresión de nombres. Lo que haremos ahora es agregar una condición adicional en donde si la edad es mayor igual a 30 y menor que 60, entonces imprima

sólo la inicial del nombre, pero si es mayor igual que 60 que imprime `La edad es mayor a 60. No se imprimirá el nombre`.

```
# Agregado una condición lógica más al ejemplo de impresión de nombres.
imp_nombre<-function(nombre,edad){
  if(edad<30){
    print(paste0("El nombre es: ",nombre))
  } else if(edad>=30 & edad<60) {
    print(paste0("La inicial del nombre es: ", substring(nombre,1,1)))
  } else {
    print("La edad es mayor a 60. No se imprimirá el nombre")
  }
}
```

Para que nos arroje la inicial del nombre estamos usando la función `substring()`[14] que nos permite extraer caracteres de un vector o elemento caracter.

Ahora veamos que nos arroja si Luis tiene 44 años.

```
imp_nombre(nombre = "Luis", edad = 44)
```

```
## [1] "La inicial del nombre es: L"
```

Como Luis tiene 44 años no se cumple la primera condición lógica, por lo que no se imprime el proceso inherente al `if`, entonces, pasa a comprobar la segunda condición lógica la cual si es `TRUE`, por lo que imprime el proceso inherente al `else if`, el cual es `"La inicial del nombre es L"`

Veamos ahora que resultaría con Antonio que tiene 87 años.

```
imp_nombre(nombre = "Antonio", edad = 87)
```

```
## [1] "La edad es mayor a 60. No se imprimirá el nombre"
```

Al tener Antonio 87 años de edad no cumple ni la primera ni la segunda condición lógica por lo que se ejecutará el proceso inherente al `else` que es: `"La edad es mayor a 60. No se imprimirá el nombre"`.

Por último, veamos el caso en el que se considere 4 escenarios. Lo que se hará es colocar otra condición en la cual si la edad es mayor a 100 entonces que el software imprima la oración `Ha vivido más de un siglo. Larga vida para usted:` concatenado con el nombre.

```
# Agregando una nueva condición.
imp_nombre<-function(nombre,edad){
  if(edad<30){
    print(paste0("El nombre es: ",nombre))
  } else if(edad>=30 & edad<60) {
    print(paste0("La inicial del nombre es: ", substring(nombre,1,1)))
  } else if(edad>=60 & edad<100){
    print("La edad es mayor a 60 pero menor que 100. No se imprimirá el nombre")
  } else {
    print(paste0("Ha vivido más de un siglo. Larga vida para: ", nombre))
  }
}
```

Veamos que sucedería con Ariadna que tiene 105 años.

```
imp_nombre(nombre = "Ariadna", edad = 105)
```

```
## [1] "Ha vivido más de un siglo. Larga vida para: Ariadna"
```

[14] Si usted no comprende esta sintaxis, le recomendamos revisar el capítulo de vectores aleatorios o buscar cuales son los argumentos de la función `sample()`.

Al no cumplirse ninguna de las 3 condiciones el software ejecutará el proceso inherente al `else` por lo que nos arroja: `"Ha vivido más de un siglo. Larga vida para: Ariadna"`.

Si usted desea agregar más escenarios, tiene todas las herramientas para lograr el objetivo.

9.4 Unicidad de la condicional if.

La unicidad de la condicional `if` se refiere a que sólo evalua a un elemento. Como está expresamente detallado en la documentación[15] de esta función.

`A length-one logical vector that is not NA. Conditions of length greater than one are currently accepted with a warning, but only the first element is used. An error is signalled instead when the environment variable _R_CHECK_LENGTH_1_CONDITION_ is set to true. Other types are coerced to logical if possible, ignoring any class.`

Nos dice que si en la condicional el vector tiene una longitud mayor a 1, entonces lo evaluará pero arrojará un mensaje de advertecia en donde indica que sólo ha evaluado al primer elemento.

Para ver esto realicemos un pequeño ejemplo.

```
# Creando el vector x.
x<-1:5

# Que imprima el vector x si es menor que 3.
if(x<3){
  print(x)
}
```

```
## Warning in if (x < 3) {: la condición tiene longitud > 1 y sólo el primer elemento será
## usado
```

```
## [1] 1 2 3 4 5
```

Como era de esperarse nos arroja el mensaje de advertencia en donde nos dice que sólo usará el primer elemento del vector `x`.

Entonces, usted puede preguntarse ¿Qué función usar cuando el vector a evaluar tiene más de un elemento?

Se tiene una variedad muy amplia de opciones, que iremos viendo con el pasar de los capítulos. Pero en el capítulo siguiente veremos un par que resuelvan este pequeño inconveniente, veremos especificamente el ciclo `for` y la función `ifelse()`.

10 LOOP FOR E IF ELSE PARA MÁS DE UN ELEMENTO.

En este capítulo se se mostrará como usar el loop `for` que nos permitirá realizar múltiples procesos con una sencilla sintaxis. Asimismo, gracias al loop `for` podremos evaluar más de un elemento al usar las condicionales `if else`. Por último, veremos el uso de la función `ifelse()` que nos otorgará el mismo resultado que si usaramos el loop `for` con las condicionales `if else`.

10.1 Loop for.

Gracias al loop `for` podremos realizar tareas similares automáticamente. El loop permite hacer el recorrido de todos los elementos de un vector con el fin de asignar a la operación procedente cada uno de los elementos del recorrido. La sintaxis del loop `for` es la siguiente:

[15] El ciclo while se desarrollará en el siguiente capítulo.

`for (var in seq) {expression}`

Donde se le asigna un iterador (`var`), que puede ser cualquier letra o palabra, que hará el recorrido por el vector (`seq`) para asignar cada elemento de este en la `expresión`.

El iterador puede jugar el papel de objeto o de subobjeto, es decir, en el primer caso el iterador entra en la expresión como un objeto que será argumento de una función, en el segundo caso el iterador entra como un subobjeto de otro objeto y en conjunto servirán de argumento de una función.

10.1.1 El iterador como objeto.

Veamos un ejemplo para que quede más claro. Vamos a imprimir el siguiente texto "Este es el número: ...", en los tres puntos asignaremos los números desde el 1 al 10.

```
# Usando el loop for.
for (i in 1:10) {
  print(paste0("Este es el número: ",i))
}
```

```
## [1] "Este es el número: 1"
## [1] "Este es el número: 2"
## [1] "Este es el número: 3"
## [1] "Este es el número: 4"
## [1] "Este es el número: 5"
## [1] "Este es el número: 6"
## [1] "Este es el número: 7"
## [1] "Este es el número: 8"
## [1] "Este es el número: 9"
## [1] "Este es el número: 10"
```

Primero, hemos asignado un iterador llamado `i` que hará el recorrido por todos los elementos del vector `1:10`, cada elemento lo usará en la ejecución de la expresión `print(paste0("Este es el número: ",i))`, es decir, el iterador `i` tomará cada uno de los elementos del vector `1:10` y los asignará en la expresión `print(paste0("Este es el número: ",i))`.

Gracias a esto hemos obtenido el resultado esperado en donde se ha impreso el texto "Este es el número: 1", "Este es el número: 2", y así sucesivamente hasta "Este es el número: 10".

Como se habrá podida dar cuenta una de las características principales del loop `for` en R es su sencilles para su uso. Se le invita a realizar la misma operación en otros softwares[16] y verá la diferencia.

10.1.2 El iterador como un subobjeto.

Ahora veamos un ejemplo en donde el vector por donde se hará el recorrido es una vector caracter.

```
# Ejemplo usando un vector caracter.
x<-c("Azul","Rojo","Amarillo","Rosado","Marrón")

for (i in 1:length(x)) {
  print(paste0("Este es el color: ",x[i]))
}
```

```
## [1] "Este es el color: Azul"
## [1] "Este es el color: Rojo"
## [1] "Este es el color: Amarillo"
```

[16] Si usted no comprende esta sintaxis, le recomendamos revisar el capítulo de vectores aleatorios o buscar cuales son los argumentos de la función `sample()`.

```
## [1] "Este es el color: Rosado"
## [1] "Este es el color: Marrón"
```

En esta ocasión hemos complicado un poco el uso, pero con el fin de que usted se de cuenta que el iterador también puede ser usado como un subobjeto, en este caso el iterador `i` es un subobjeto del objeto `x`.

Primero, se ha definido el vector `x` con los nombres: "Azul","Rojo","Amarillo","Rosado","Marrón". Posteriormente, en la sintaxis del loop `for` se ha asignado un iterador llamado `i` que hará el recorrido por el vector `1:length(x)`, es decir, desde el número 1 hasta el tamaño del vector `x` (desde 1 al 5). Cada uno de estos elementos se asignarán en la expresión `x[i]` (llamará a cada uno de los elementos del vector `x`, desde el primero al último, ya que se convertirá en `x[1]`, `x[2]`, `x[3]`, `x[4]` y `x[5]`).

Lo principal de este ejemplo, es darse cuenta de la importancia que juega el iterador que ya no solo juega el papel de objeto, como en el primer ejemplo, sino como subobjeto de otro objeto.

Quizá le parezca un poco complicado al inicio, pero como se comentó al inicio este loop puede ser escrito de una manera más sencilla, a continuación, se muestra la sintaxis.

```
# El ejemplo 2 con una sintaxis más sencilla.
for (i in x) {
  print(paste0("Este es el color: ",  i))
}
```

```
## [1] "Este es el color: Azul"
## [1] "Este es el color: Rojo"
## [1] "Este es el color: Amarillo"
## [1] "Este es el color: Rosado"
## [1] "Este es el color: Marrón"
```

En este caso se puede observar que el iterador juega el papel de objeto. ¿Cómo así? El iterador `i` recorrerá el vector `x`, es decir, todos los elementos de `x` ("Azul","Rojo","Amarillo","Rosado","Marrón") y los asignará en la expresión `print(paste0("Este es el color: ",  i))`.

Usted se puede estar preguntando ¿Cuál es el fin de colocar al iterador como subobjeto si la sintaxis es más sencilla considerando al iterador como objeto? En el ejemplo que hemos visto y por su sencilles es más fácil usar al iterador como objeto, pero cuando avancemos más y deseemos hacer loops más complicados en donde haya más de un iterador o dentro de funciones, tendremos que usar al iterador como subobjeto. Así que por el momento es preciso aprender las dos formas.

Avancemos un poco más y realicemos un ejemplo un poquillo más complicado.

Vamos a realizar un loop en el cual se guardan los resultados en un vector externo.

```
# Guardando los resultados en un vector externo.
x<-c()
for (t in 1:5) {
  y<-t^2
  x<-c(x,y)
}
```

Primero hemos creado un vector vacio `x` donde guardaremos los resultados del loop, posteriormente en el loop `for` se asigna el iterador `t` que hará el recorrido por el vector `1:5`, asignando cada elemento en la expresión `y<-t^2; x<-c(x,y)`. Lo que hará será elevar los elementos del vector `1:5` al cuadrado y guardarlo en el objeto `y`. En la expresión `x<-c(x,y)` le decimos al software que guarde en el objeto `x` el vector que se genera al juntar el objeto `x` con el objeto `y`. Como el valor de `y` cambiará con cada elemento ya que el iterador tomará uno por uno los elementos del vector `1:5`, entonces, es preciso guardar cada resultado.

Para observar que se ha guardado en el objeto `x` corramos este vector.

```
x
```

```
## [1]  1  4  9 16 25
```

Como se puede dar cuenta el vector `x` tiene los elementos 1, 4, 9, 16, 25 que resultan al elevar cada elementos del vector `1:5` al cuadrado.

Usted puede ser perpicaz y al haber leído todos los capítulos de este libro, usted puede decir que el resultado anterior saldría solo con ejecutar `(1:5)^2`. En efecto, sí, el resultado es el mismo, pero repito, por el momento estamos haciendo ejemplos muy sencillos para que se tenga una total comprensión de los usos del loop `for`.

10.2 El loop for con las condicionales if else.

Usted recuerda del capítulo anterior la unicidad de las condicionales `if else`, el cual implicaba que solo se evaluara el primer elemento de un vector. Ahora gracias al loop `for` podremos evaluar todos los elementos del vector.

Veamos un ejemplo en donde se da a conocer el problema y cómo podríamos solucionar esto usando el loop `for`.

El ejemplo es el siguiente, queremos saber que elementos del vector `p` son menores a 30, y si es así que imprima los elementos. Es obvio que tenemos que usar la condicional `if`.

```
# Creando el vector p.
set.seed(3000)
p<-sample(20:50, 20, T)

# El vector p tiene los elementos.
p
```

```
##  [1] 42 49 34 26 34 44 29 30 40 42 38 30 36 22 42 21 46 21 28 34
```

```
# Realizando la condicional if.
if(p<30){
  print(p)
}
```

```
## Warning in if (p < 30) {: la condición tiene longitud > 1 y sólo el primer elemento será
## usado
```

¿Cómo es posible que no haya impreso ningún elemento? Como se mencionó, la condicional `if` sólo evaluó el primer elemento del vector `p` y este al no ser 42 menor que 30, nos resulta que la prueba lógica `p<30` es `FALSE` por lo cual no ejecuta el proceso `print(p)`.

Entonces, estamos atados de manos, pero usted puede observar una posible solución usando la siguiente sintaxis:

```
if(p[1]<30){
  print(p)
}

if(p[2]<30){
  print(p)
}

if(p[3]<30){
  print(p)
}

# Así sucesivamente hasta
```

```
if(p[20]<30){
  print(p)
}
```

Pero esto no es para nade eficiente, y sería un demente si pierde su tiempo escribiendo esa sintaxis.

Entonces, nuestra salvación es el loop `for`. A continuación se muestra la sintaxis usando al iterador como un objeto.

```
# Resolviendo el problema usando el loop for.
for (i in p) {
 if(i<30){
  print(i)
}
}
```

```
## [1] 26
## [1] 29
## [1] 22
## [1] 21
## [1] 21
## [1] 28
```

Y a continuación la solución usando al iterador como un subobjeto.

```
# Resolviendo el problema usando el loop for.
for (i in 1:length(p)) {
 if(p[i]<30){
  print(p[i])
}
}
```

```
## [1] 26
## [1] 29
## [1] 22
## [1] 21
## [1] 21
## [1] 28
```

Con estas soluciones se puede observar el tiempo que nos hemos ahorrado si sabemos usar correctamente el loop `for`.

Ahora veamos otro ejemplo en donde tenemos el vector sexo que tiene 30 elementos que pueden ser 0 y 1. Asimismo, nos indican que 0 indica que el sexo es femenino y que 1 indica que el sexo es masculino. Entonces, nos piden que cambiemos los 0 y 1 por los nombres femenino y masculino.

Con los conocimientos que tenemos podemos aplicar el loop for de la siguiente manera.

```
# creando el vector sexo.
set.seed(2021)
sexo<-sample(0:1, 30, T)

# Los elementos del vector sexo.
sexo
```

```
##  [1] 0 1 1 1 0 1 1 1 1 1 0 1 1 0 0 0 0 0 1 0 1 1 0 1 0 0 1 1 1 1
```

```
# Aplicando el loop for.
for (i in 1:length(sexo)) {
```

```
  if(sexo[i]==0){
    sexo[i]<-"Femenino"
  } else if (sexo[i]==1){
    sexo[i]<-"Masculino"
  }
}
```

Estamos usando el método donde se considera al iterador como subobjeto ¿Le invitamos a realizar el mismo ejericio pero usando el iterador como objeto? ¿Con qué método le resultó más fácil?

Viendo como quedó el vector `sexo` luego de aplicar el loop `for`.

```
sexo
```

```
##  [1] "Femenino"  "Masculino" "Masculino" "Masculino" "Femenino"  "Masculino" "Masculino"
##  [8] "Masculino" "Masculino" "Masculino" "Femenino"  "Masculino" "Masculino" "Femenino"
## [15] "Femenino"  "Femenino"  "Femenino"  "Femenino"  "Masculino" "Femenino"  "Masculino"
## [22] "Masculino" "Femenino"  "Masculino" "Femenino"  "Femenino"  "Masculino" "Masculino"
## [29] "Masculino" "Masculino"
```

En efecto, todos los elementos que eran 0 se han cambiado por "Femenino" y todos los que era 1 se han cambiado por "Masculino".

Estoy seguro que usted se está preguntando ¿Todo eso para cambiar los 0 por "Femenino" y los 1 por "Masculino"? En realidad sí. ¿Pero si con otros softwares se obtiene el mismo resultado muchó más fácil? En realidad sí. Pero repito, aquí estamos viendo las formas de usar el loop `for` con la condicional `if else`. Esto que parece innecesario, a la hora de aplicarlo a matrices, listas o dataframe será tan útil y verá la facilidad con lo que resuelve futuras cuestiones. Veremos estas aplicaciones más útiles cuando completemos el tema de data frame, por el momento, será suficiente con conocer como se aplica el loop con sus dos variantes. Pero sí ha quedado insatisfecho, a continuación, se muestra como resolver este tipo de cuestiones usando la función `ifelse()`.

10.3 Función ifelse().

Una forma de realizar el proceso anterior sin la necesidad de usar el loop `for` explícitamente y sin tanto código, es usar la función `ifelse()`, el cuál realiza el loop `for` implícitamente y resulta muchísimo más sencillo de usar. Pero como todo proceso que genere simplicidad, solo servirá para un número pequeño de aplicaciones, por lo cual, no englobará en al loop `for`.

Vamos a realizar el mismo ejercicio que el anterior.

```
# creando el vector sexo.
set.seed(2021)
sexo<-sample(0:1, 30, T)

# Los elementos del vector sexo.
sexo
```

```
##  [1] 0 1 1 1 0 1 1 1 1 1 0 1 1 0 0 0 0 0 1 0 1 1 0 1 0 0 1 1 1 1
```

```
# Resolviendo con la función ifelse.
ifelse(sexo==0,"Femenino","Masculino")
```

```
##  [1] "Femenino"  "Masculino" "Masculino" "Masculino" "Femenino"  "Masculino" "Masculino"
##  [8] "Masculino" "Masculino" "Masculino" "Femenino"  "Masculino" "Masculino" "Femenino"
## [15] "Femenino"  "Femenino"  "Femenino"  "Femenino"  "Masculino" "Femenino"  "Masculino"
## [22] "Masculino" "Femenino"  "Masculino" "Femenino"  "Femenino"  "Masculino" "Masculino"
## [29] "Masculino" "Masculino"
```

Y en efecto, hemos obtenido el mismo resultado.

¿Cómo se hizo?

La función `ifelse()` tiene la siguiente sintaxis.

```
ifelse(test, yes, no)
```

En donde el primero argumento `test` es una prueba lógica que puede ser TRUE o FALSE; el segundo argumento `yes` es el valor que retornará la función si el `test` resulta TRUE; por último, el argumento `no` es el valor que retornará la función si el `test` resulta FALSE.

En ese sentido, la prueba lógica o `test` en nuestro ejemplo fue `sexo==0`, en `yes` hemos asignado el valor de "Femenino" y en `no` el valor de "Masculino", lo que implica que si la variable `sexo==0` es TRUE, entonces asignará "Femenino", pero si es FALSE asignará "Masculino".

El uso de esta función es muy sencilla y puede anidarla. Para observar un pequeño ejemplo supongamos que tenemos el vector civil, el cual tiene elementos entre 0, 1 y 2. Y nos indican que 0 es soltero, 1 casado y 2 viudo.

```
# Creando el vector civil.
set.seed(2021)
civil<-sample(0:2, 20, T)

# Viendo los elementos del vector civil.
civil
```

```
##  [1] 2 1 1 1 2 1 1 2 1 1 0 2 2 0 2 2 1 2 0 2
```

```
# Resolviendo el problema usando la función ifelse.
ifelse(civil==0, "Soltero",
       ifelse(civil==1, "Casado", "viudo"))
```

```
##  [1] "viudo"   "Casado"  "Casado"  "Casado"  "viudo"   "Casado"  "Casado"  "viudo"
##  [9] "Casado"  "Casado"  "Soltero" "viudo"   "viudo"   "Soltero" "viudo"   "viudo"
## [17] "Casado"  "viudo"   "Soltero" "viudo"
```

En efecto, hemos obtenido el resultado esperado. Cuando se vea los capítulos del paquete `dplyr` se mostrará el uso de la función `case_when()` que realiza el mismo procedimiento que la función `ifelse()`, pero con una menor sintaxis y con código más limpio y entendible.

10.4 Declaración Break y Next.

En R se tiene las declaraciones `break` y `next`. La declaración `break` nos permite detener el loop `for` o el ciclo `while`[17], mientras que la declaración `next` nos permite saltar una deteminada iteración en el loop `for`. Veamos a continuación la aplicación de cada uno.

Como se mencionó la declaración `break` detiene el loop `for` siempre y cuando una determinada condición lógica sea `TRUE`. Por ejemplo, vamos a imprimir las letras del abecedario hasta la letra H. Una vez que se ha impreso la letra H que se detenga el loop `for`.

Para esto vamos a usar la función `LETTERS` que nos arroja las letras del abecedario en mayúscula.

```
# Las letras del abecedario en mayúscula.
LETTERS
```

```
##  [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L" "M" "N" "O" "P" "Q" "R" "S" "T" "U"
## [22] "V" "W" "X" "Y" "Z"
```

[17] El ciclo while se desarrollará en el siguiente capítulo.

```r
# Construyendo el loop for.

for (i in LETTERS) {
  if(i=="H"){
    print(i)
    break
  } else {
   print(i)
  }
}
```

```
## [1] "A"
## [1] "B"
## [1] "C"
## [1] "D"
## [1] "E"
## [1] "F"
## [1] "G"
## [1] "H"
```

En efecto, solo se ha impreso hasta la letra "H". Pero ¿cómo sucedió? Primero, si nosotros no consideramos a la declaración `break` el loop `for` hubiese impreso todas las letras del abecedario. Entonces, al incluir el `break` le decimos al software que cuando el iterador sea igual a "H" entonces imprima la letra de la iteración y que detenga el loop. Ahora, si no se cumple la condión, entonces que imprima la letra de la iteración. Es así que cuando el iterador vale "H" se cumple la condición entonces imprime "H" y detiene el loop.

Por otro lado, la declaración `next` nos permite saltar la iteración siempre y cuando la condición lógica sea `TRUE`. Veamos un ejemplo para ver su aplicación.

Por ejemplo, vamos a ser el mismo loop que en el ejemplo anterior, pero ahora vamos a saltar la iteración, siempre y cuando la iteracción sea igual a la letra "H". Es decir, si la iteración es igual a "H" entonces no será impreso.

```r
for (i in LETTERS) {
  if(i=="H"){
    next
  } else {
   print(i)
  }
}
```

```
## [1] "A"
## [1] "B"
## [1] "C"
## [1] "D"
## [1] "E"
## [1] "F"
## [1] "G"
## [1] "I"
## [1] "J"
## [1] "K"
## [1] "L"
## [1] "M"
## [1] "N"
## [1] "O"
## [1] "P"
```

```
## [1] "Q"
## [1] "R"
## [1] "S"
## [1] "T"
## [1] "U"
## [1] "V"
## [1] "W"
## [1] "X"
## [1] "Y"
## [1] "Z"
```

Y en efecto, no se ha impreso la letra "H" y esto sucede porque en la sintaxis indicamos que si el iterador es igual a "H" entonces ejecute `next`, es decir, no imprima a la letra "H".

En el siguiente capítulo veremos el tema del ciclo `while()` que es complementario al ciclo `for()`. Este ciclo, también usa las declaraciones `next` y `break`, incluso, su aplicación es más útil que en el loop `for()`.

11 CICLO WHILE.

En este capítulo aprenderá sobre el uso y aplicaciones del ciclo `while()`. Primero, se verá la sintaxis del ciclo, posteriormente unos ejemplos siemples y otros usando las declaraciones `break` y `next`; y en la parte final, su aplicación en economía, específicamente en la maximización de beneficios del productor.

11.1 Sintaxis del ciclo while.

El ciclo `while()` sirve para ejecutar una determinada expresión siempre y cuando la condición lógica sea `TRUE`, una vez que la condición lógica pasa a ser `FALSE` el ciclo finaliza. Es así que nos servirá para ejecutar expresiones secuenciales.

La sintaxis del ciclo `while()` es la siguiente.

```
while (cond) {expression}
```

Donde `cond` es la condición lógica y `expression` es la expresión o expresiones que se ejecutarán siempre y cuando la condición lógica sea `TRUE`.

Ahora veamos unos ejemplos de como usar este ciclo.

11.2 Ejemplos de uso del ciclo while.

como primer ejemplo vamos a imprimir la oración "EL objeto z vale: ..." (en los 3 puntos irá el valor que toma z) siempre y cuando el objeto `z` que vale 8 sea menor que 15. Si el objeto `z` es mayor o igual que 15, entonces, se dejará de imprimir la oración. Asimismo, por cada iteración el valor de `z` se va incrementando en una unidad.

Veamos como sería la sintaxis para este ejemplo.

```
# Definiendo el objeto z.
z<-8

# Aplicando el ciclo while.
while (z<15) {
  print(paste0("El objeto z vale: ",z))
  z<-z+1
}
```

```
## [1] "El objeto z vale: 8"
## [1] "El objeto z vale: 9"
## [1] "El objeto z vale: 10"
## [1] "El objeto z vale: 11"
## [1] "El objeto z vale: 12"
## [1] "El objeto z vale: 13"
## [1] "El objeto z vale: 14"
```

En efecto solo se ha impreso la oración hasta que el valor de `z` sea menor a 15. Pero ¿cómo sucedió? Primero, se ha creado el objeto `z` que es igual a 8, luego al crear el ciclo `while()` en la condición escribimos `z<15`, por lo que para cada iteración el software evaluaba si `z` era menor a 15 y si el resultado era `TRUE` entonces ejecutaba las expresiones: `print(paste0("El objeto z vale: ",z))` y `z<-z+1`. En el primer caso imprimía la oración y en el segundo sumaba 1 al valor de `z`. Entonces, al inicio `z` valía 8 entro en la condición y `z` era menor que 15 por lo que la condición era `TRUE` entonces ejecutó las expresiones: "El objeto z vale: 8" y sumó a `z` el valor de 1 con lo cual `z` pasó a valer 9. Continuó con el mismo proceso hasta que `z` fue igual a 15 en donde al probar la condición lógica `z<15` resultó `FALSE` por lo que detuvo el ciclo.

Gracias al ciclo `while()` se puede realizar procesos infinitos, para correr un bucle infinito solo tenemos que colocar `TRUE` en la condición lógica y ejecutará la expresión infinitas veces (esto no es al 100% porque se detendrá el proceso cuando el computador no pueda seguir ejecutando la expresión, por problemas de capacidad).

A continuación se muestra la sintaxis para un bucle infinito, que imprima infinitas veces la oración "Detenme por favor". Le recomendamos no correr esta sintaxis, pero si lo hace, para cancelar el bucle solo tiene que presionar la tecla "esc" (escape) en su computador.

```
# Creando el bucle infinito.
while (TRUE) {
  print("Detenme por favor")
}
```

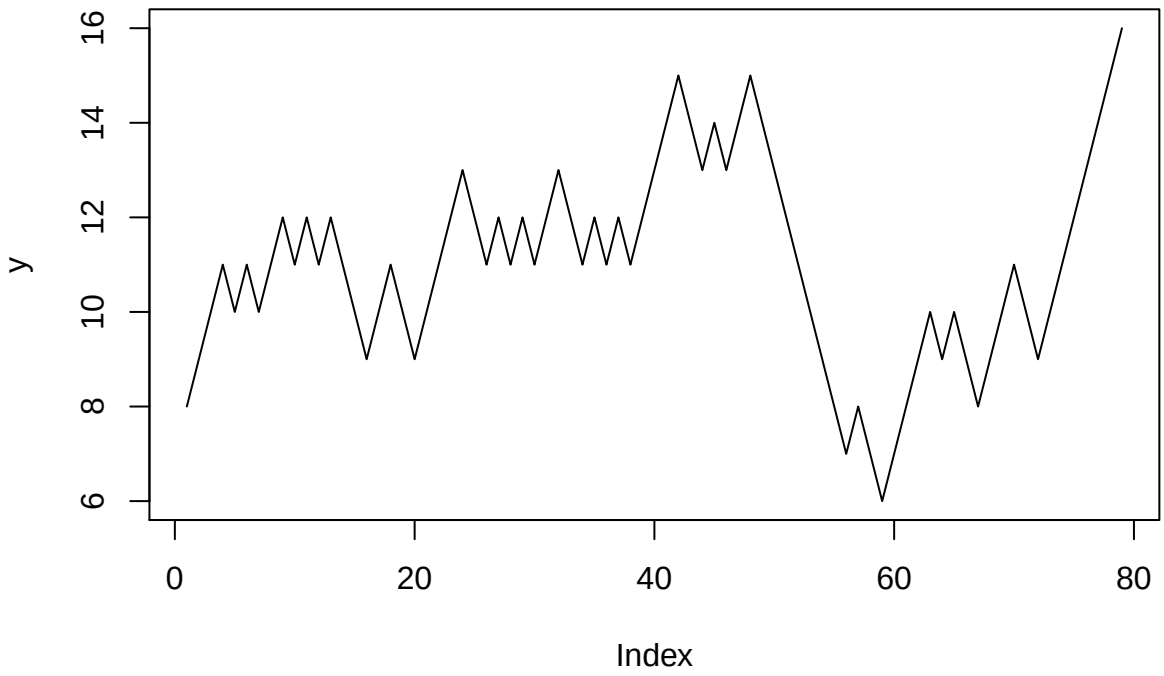
Si usted lo ejecuta, verá que se imprime la oración "Detenme por favor" muchas veces hasta que usted detiene el proceso. Pero ¿Cómo sucedio? Primero, lo que hemos hecho es colocar `TRUE` en la condición, lo cual hace que la condición lógica siempre sea `TRUE` por lo que el ciclo `while()` no se detendrá, entonces, al ser `TRUE` la condición ejecutará la expresión `print("Detenme por favor")` hasta que usted lo detenga.

Ahora veremos un ejemplo usando elementos aleatorios. Vamos a definir el objeto `x` y con valor igual a 7, a este objeto se le irá sumando aleatoriamente `+1` o `-1` y se realizará este proceso hasta siempre y cuando el objeto `x` sea mayor igual que 0 y menor igual a 15. Y como último paso graficamos los valores que ha tomado `x` hasta que la condición lógica sea `FALSE`.

```
# Creando el objeto x.
x<-7

# Creando el vector y.
y<-c()

# El ciclo while.
while (x>=0 & x<=15) {
  x<-x+sample(c(-1,1), 1)
  y<-c(y,x)
}

# Viendo el último valor que toma x
x
```

```
## [1] 16
```

```r
# Por último graficamos los valores que ha tomado x.
plot(y, type = "l")
```



Lo que hemos construido es una caminata aleatoria (random walk), y el gráfico lo representa de una manera muy clara. Expliquemos lo que se ha realizado. Primero, se ha creado el objeto `x` que es igual a 7 y el objeto `y` que es un vector vacío (Estamos creando este vector porque ahí se guardarán los valores que tomará x). Luego, contruimos el `while()`, en la condición colocamos las condiciones lógicas `x>=0 & x<=15`, entonces, mientras `x` cumpla esas condiciones se ejecutarán las expresiones `x<-x+sample(c(-1,1), size = 1)` y `y<-c(y,x)`. La primera expresión es la caminata aleatoria, ya que tenemos un valor inicial determinado (`x==7`) y un proceso aleatorio (`sample(c(-1,1), 1)`)[18]. Entonces, cada vez que se ejecuta las expresiones se suma un valor igual a +1 o -1 a `x` y esto se guarda en el objeto `y`. El ciclo continuará hasta que x no cumpla la condición lógica.

Asimismo, vemos que se detiene el ciclo, porque `x` ya no cumple la condición lógica, ya que `x` es igual a 16. Por último estamos graficando el vector `y`, ya que en ese vector se han guardado los distintos valores que toma `x`. Para esto usamos la función `plot()` que se verá más a detalle en capítulos posteriores.

Ahora, veamos un ejemplo similar, pero ahora tendremos dos objetos, el objeto `m` y el objeto `p` que valdrán 4 cada uno. A estos objetos se les sumará `+1` o `-1` hasta que `m` sea mayor igual que 0 o `p` sea menor igual que 10. Y como último paso graficamos los valores que ha tomado `m` y `p`.

```r
# Definiendo los objetos m y p.
m<-4
p<-4

# Definiendo los objetos en donde se guardarán los resultados.
```

[18] Si usted no comprende esta sintaxis, le recomendamos revisar el capítulo de vectores aleatorios o buscar cuales son los argumentos de la función `sample()`.

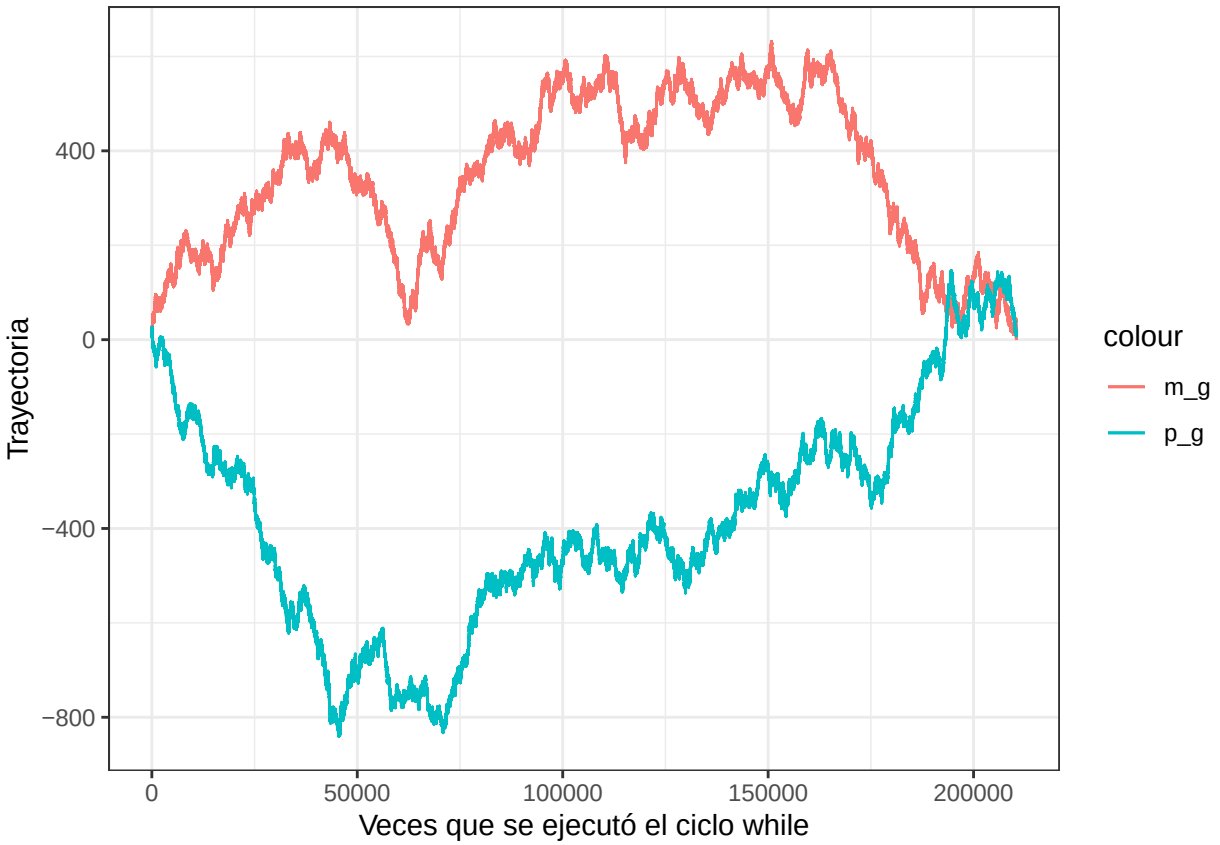
```r
m_g<-c()
p_g<-c()

# El ciclo while.
while (m>=0 | p<=10) {
  m<-m+sample(c(1,-1), 1)
  p<-p+sample(c(1,-1), 1)
  m_g<-c(m_g,m)
  p_g<-c(p_g,p)
}

# Los valores finales que toman m y p.
m
```

```
## [1] -1
```

```r
p
```

```
## [1] 13
```

```r
# Graficando la trayectoria de m y p.
library(ggplot2)

df<-data.frame(m_g,p_g)
df$n<-1:length(df$m_g)

ggplot(df,aes(x=n,y=m_g, color="m_g"))+
  geom_line()+
  geom_line(aes(y=p_g, color="p_g"))+
  labs(x="Veces que se ejecutó el ciclo while",
       y="Trayectoria")+
  theme_bw()
```



Igual que en el ejemplo anterior hemos creado 2 caminatas aleatorias, la diferencia es que ahora en la condición lógica se especificó el operador o `|` con lo cual para que la condición sea `FALSE` los dos objetos tenían que estar fuera de la condición, por un lado `m` tenía que ser menor que 0 y a la vez `p` tenía que ser mayor que 8. Esto se comprueba, ya que `m` cuando se detuvo el ciclo vale -1 y `p` vale 13.

Por último, estamos graficando las trayectorias pero en este caso estamos usando el paquete `ggplot2`, el uso de este paquete también se desarrollará en capítulos posteriores. Por el momento, es suficiente si se ha llegado a comprender el uso del ciclo `while()`.

Recuerde que cuando usted replique este ejemplo y el anterior obtendrá resultados distintos, y por ende, gráficos distintos ya que estamos ejecutando procesos aleatorios.

11.3 Aplicaciones con las declaraciones Break y Next.

Las declaraciones `break` y `next` cumplen el mismo rol que se explicó en la capítulo anterior. Es así que si deseamos detener el ciclo `while()` por más que la condición lógica sea `TRUE` tendremos que usar `break` y si deseamos saltar una iteración, entonces usaremos la declaración `next`.

Veamos primero un ejemplo de como usar la declaración `break`. Vamos a crear una ecuación de segundo grado y vamos asignar valores continuamente hasta que se encuentre una de las raíces. Como no sabemos en que rango se encontrará la raíz, entonces haremos un bucle infinito y se detendrá cuando encuentre la solución.

```
# La ecuación a encontrar solución.
# x^2+2*x-3=0

# EL ciclo while.
```

```
x<-0
while (TRUE) {
  y<-x**2+2*x
  if(round(y,3)==3){
    print(paste0("La raíz o solución de la ecuación es: ", x))
    break
  }
  x<-x+0.001
}
```

```
## [1] "La raíz o solución de la ecuación es: 1"
```

Nos arroja `"La raíz o solución de la ecuación es: 1"` y en efecto es 1. Es fácil comprobar por el método de factorización por aspa simple que las raíces son 1 y -3. Pero ¿Qué se hizo? Primero se definió el objeto `x` igual a 0. Posteriormente en la condicional del ciclo se colocó `TRUE` por lo que el bucle es infinito, por lo que ejecutará `y<-x**2+2*x` y se detendrá cuando el valor aproximado hasta las milésimas sea igual a 3, si eso esucede entonces imprimirá "La raíz o solución de la ecuación es: ..." y se detendrá el ciclo, pero si no es así entonces, al valor de `x` le irá sumando 0.001 en cada iteración. Es así que cuando x fue igual a 1 el valor de `y` fue 3. Por lo que se detuvo el ciclo.

Estoy seguro que usted se esté preguntando ¿tengo que hacer todo eso para saber las raíces de las ecuaciones? ¿Y si no sé desde dónde parte? ¿Y si le sumo a x valores más pequeños? Y otras dudas. Déjeme decirle que este sólo era un ejemplo de uso del ciclo `while()` con la declaración `break` y por eso fue programado de esa forma, pero si su interés es calcular las raíces de las ecuaciones cuadráticas o de mayor exponente, hay otras formas más eficientes de obtenerlo. Para una muestra se deja a continuación el siguiente código que resuelve el mismo problema.

```
# Método sencillo para calcular las raíces de ecuaciones.
as.numeric(polyroot(c(-3,2,1)))
```

```
## [1]  1 -3
```

Y en efecto nos resulta 1, -3. ¿Le pareció interesante? Estoy seguro que sí, pero no coma ansias este tipo de soluciones y otras se verán en capítulos posteriores.

Ahora veamos un ejemplo en donde se use la declaración `next`, vamos a sumarle 1000 a cada uno de los elementos del vector `t`, pero si estos elementos son múltiplos de 5 no se les sumará nada.

```
# Definiendo el vector t.
set.seed(2021)
t<-sample(1:50, 20)
t
```

```
##  [1]  7 38 46 39 12  6 49 44  5 47 23 48 18  3 26 22 31 19  4 21
```

```
# El ciclo while.
r<-0
while(r<=length(t)-1){
  r<-r+1
  if(t[r]%%5==0){
    next
  } else{
    t[r]<-t[r]+1000
  }
}

# Cómo quedó el vector t?
t
```

```
##  [1] 1007 1038 1046 1039 1012 1006 1049 1044    5 1047 1023 1048 1018 1003 1026 1022 1031
## [18] 1019 1004 1021
```

Y en efecto, solo los números que no son múltiplo de 5 se les ha sumado 1000. Pero ¿Cómo sucedió? Primero creamos el vector `t` que tiene 20 elementos aleatorios. Luego creamos el objeto `r` igual a 0, este objeto nos permitirá hacer el recorrido por todos los elementos del vector `t`, será como el iterador. Luego en la condicional `r<=length(t)-1` le indicamos al software que ejecute las expresiones siempre y cuando el valor de `r` sea menor o igual al tamaño del vector `t` menos 1. Entonces, sí el elemento es múltiplo de 5 `t[r]%%5==0` pasará a la siguiente iteración, sino es así le sumará 100.

11.4 Aplicación en la maximización de beneficios del productor.

Ahora, veamos una aplicación a la economía, específicamente en la teoría de la empresa. Vamos a maximizar los beneficios del productor, para esto tenemos que recordar que los productores maximizan sus beneficios cuando los ingresos marginales son iguales a los costos marginales.

Sabemos que la función de precios del productor viene determinada por:

$$P = 600 - 4Q$$

Y la función de costos totales por:

$$CT = Q^2 + 8000$$

El productor actualmente está produciendo 10 unidades y quiere saber cómo sería la trayectoria de las utilidades si produciría hasta una cantidad de 100, asimismo desea saber cuál es la producción que maximiza sus beneficios, así como sus ingresos y costos totales en ese nivel de producción.

Para resolverlo tenemos que recordar que si la diferencia entre el ingreso marginal y el costo marginal es cero entonces el productor está maximizando la utilidad, entonces, como primer paso debemos de calcular el ingreso marginal y el costo marginal.

Calculando el ingreso marginal.

$$IT = P * Q - - - - > IT = (600 - 4Q)Q$$
$$IT = 600Q - 4Q^2$$
$$IMg = 600 - 8Q$$

Calculando el costo marginal.

$$CT = Q^2 + 8000$$
$$CMg = 2Q$$

Una vez que tenemos los ingresos y costos marginales. Procedemos a calcular lo que se pide en el ejercicio.

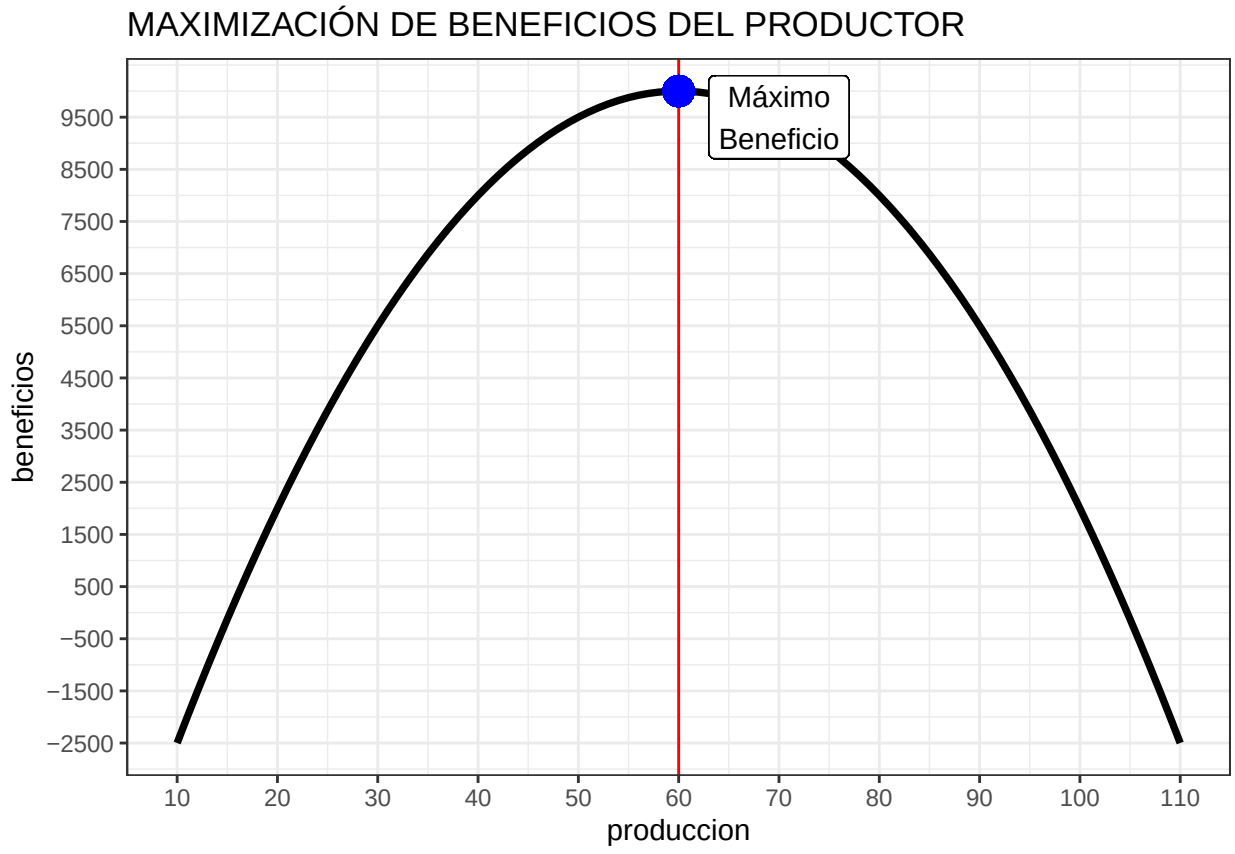
```
produccion<-10
beneficios<-c()

while (produccion<=110) {
  IMg<-600-8*produccion
  CMg<-2*produccion
  IT<-600*produccion-4*produccion^2
  CT<-produccion^2+8000
  beneficios<-c(beneficios,IT-CT)
  if(IMg-CMg==0){
```

```
    print(paste0("La producción máxima se alcanza cuando se produce: ",
                 produccion," unidades"))
  }
  produccion<-produccion+1
}
```

```
## [1] "La producción máxima se alcanza cuando se produce: 60 unidades"
```

El software nos indica que la producción que maximiza los beneficios es de 60 unidades producidas. Podemos graficar este resultado.

```
df<-data.frame(produccion=10:110,
               beneficios=beneficios)

library(ggplot2)
library(ggrepel)

ggplot(df, aes(x=produccion,y=beneficios))+
  geom_line(size=1.3)+
  scale_x_continuous(breaks = seq(10,110,10))+
  scale_y_continuous(breaks = seq(min(beneficios),max(beneficios),1000))+
  geom_vline(xintercept = 60, color="red")+
  geom_label(x=70, y=max(beneficios)-500,label="Máximo\nBeneficio")+
  geom_point(x=60,y=max(beneficios), size=5, color="blue")+
  labs(title = "MAXIMIZACIÓN DE BENEFICIOS DEL PRODUCTOR")+
  theme_bw()
```



Se puede apreciar que el máximo beneficio se alcanza con la producción de 60 unidades.

Al igual que la vez anterior, el gráfico se hizo con `ggplot2`, que se verá en capítulos posteriores.

Ahora, para calcular los ingresos y costos totales que maximizan el beneficio podemos correr lo mismo pero solo agregamos un `brake` cuando el IMg es igual al CMg.

```
produccion<-10
beneficios<-c()

while (produccion<=110) {
  IMg<-600-8*produccion
  CMg<-2*produccion
  IT<-600*produccion-4*produccion^2
  CT<-produccion^2+8000
  beneficios<-c(beneficios,IT-CT)
  if(IMg-CMg==0){
    print(paste0("La producción máxima se alcanza cuando se produce: ",
                 produccion," unidades"))
    break
  }
  produccion<-produccion+1
}
```

```
## [1] "La producción máxima se alcanza cuando se produce: 60 unidades"
```

Lo que nos resulta en los montos de ingresos y costos totales cuando se alcanza la maximización de beneficios.

```
# Ingresos totales.
IT
```

```
## [1] 21600
```

```
# Costos Totales.
CT
```

```
## [1] 11600
```

```
# Beneficio total.
beneficios[length(beneficios)]
```

```
## [1] 10000
```

Por último, solo como un aperitivo para lo que vendrá en los siguientes capítulos. Es posible que usted haya olvidado derivar o su función de costo o ingresos es muy complicada de derivar. No se preocupe que el software R también deriva e integra.

Para el ejercicio, si no sabriamos derivar, no sabríamos como se calcula el ingreso marginal y el costo marginal, entonces se lo dejaríamos al software.

```
# Para calcular el ingreso marginal.
IT<-expression(600*Q-4*Q^2)

# Derivando.
D(IT,"Q")
```

```
## 600 - 4 * (2 * Q)
```

```
# Para calcular el costo marginal.
CT<-expression(Q^2+8000)
```

```r
# Derivando.
D(CT,"Q")
```

```
## 2 * Q
```

Si usted regresa líneas atrás podrá darse cuenta que esos son los valores que se usaron para el ingreso y costo marginal. Igualmente, por el momento es suficiente con que quede claro el uso del ciclo `while()` en capítulos posteriores se mostrará como derivar e integrar para hacer los procesos más automáticos.

En el siguiente capítulo veremos el tema de data frame. Uno de los objetos primordiales cuando se desea trabajar con datos.