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

César Anderson Huamaní Ninahuanca

2021-02-04

Contents

Prólogo **1**

1 R COMO CALCULADORA. **2**
1.1 PRINCIPALES OPERACIONES MATEMÁTICAS. 2
1.2 DECIMALES EXACTOS Y REDONDEO DE CIFRAS. 5
1.3 GUARDAR O DEFINIR OBJETOS. 7
1.4 FUNCIONES. 8

2 VECTORES. **10**
2.1 DEFINICIÓN DE UN VECTOR. 10
2.2 CLASES DE VECTORES. 10
2.3 NÚMERO DE ELEMENTOS DE UN VECTOR. 14
2.4 FUNCIONES PARA CONSTRUIR VECTORES MÁS EFICIENTEMENTE. 14
2.5 CONCATENAR ELEMENTOS Y VECTORES. 18

3 OPERACIONES MATEMÁTICAS Y ESTADÍSTICAS CON VECTORES. **19**
3.1 OPERACIONES MATEMÁTICAS. 19
3.2 OPERACIONES ESTADÍSTICAS. 21

4 VECTORES ALEATORIOS y SELECCIÓN DE ELEMENTOS. **25**
4.1 CREACIÓN DE VECTORES CON ELEMENTOS ALEATORIOS. 25
4.2 SELECCIÓN DE ELEMENTOS. 29

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
```

[1] Es probable que a usted le salga distintos números ya que el algoritmo de selección depende de la versión del software. Este libro se desarrolló usando la versión `4.0.3` de R.

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 `%%`.

```
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.

```
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 `**`.

```
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.

```
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()`.

```
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()`.

```
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)
```

[2]Esto no implica que nos salga los mismos resultados que anteriormente, por más que se ha colocado la misma semilla, los argumentos de la sintaxis anterior han variado. Lo cual implica un nuevo proceso aleatorio.

```
## [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
```

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))
```

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

```
## [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.

```
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
```

[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.

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.

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`.

[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.

```r
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:

```r
# 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 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.

```r
# 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.

```r
# 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)
```

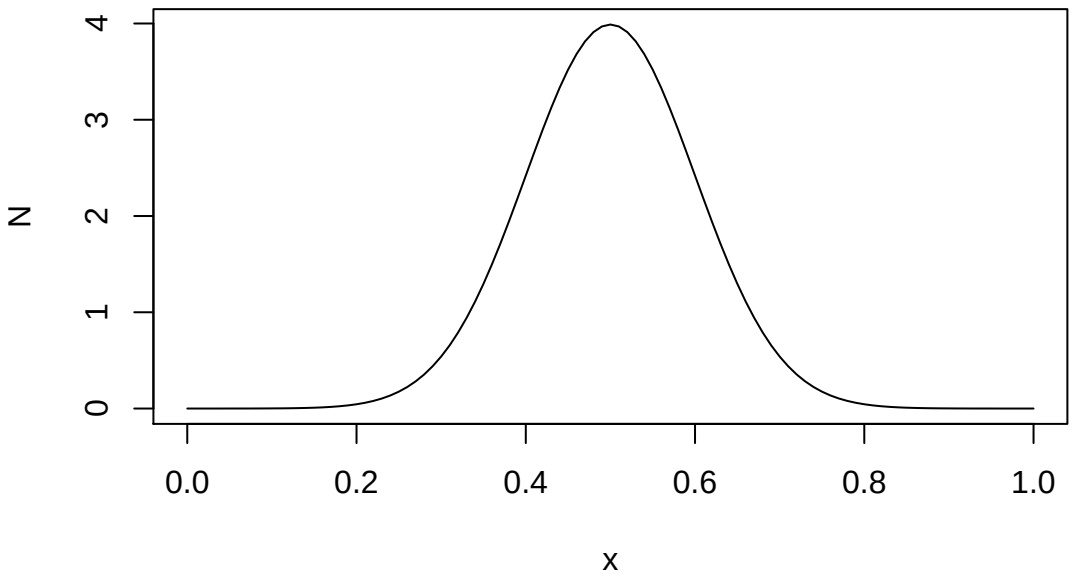
```
## [1] 0.04431848
```

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.

```r
# 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
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
```

[6]Es probable que a usted le salga distintos números ya que el algoritmo de selección depende de la versión del software. Este libro se desarrolló usando la versión `4.0.3` de R.

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:

```
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.

```
# 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.

```
# 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
```

```
# 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
```

```
# 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 10 10  1  1
## [37]  3  3  3  4  4  4  5  5  5  6  6  6  7  7  7  8  8  8  9  9  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:

```
# 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:

```
# 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.

```
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.

```
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.

```
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.

```
# 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 44.555556 50.000
```

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.

```
# 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:

```
# 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] Esto no implica que nos salga los mismos resultados que anteriormente, por más que se ha colocado la misma semilla, los argumentos de la sintaxis anterior han variado. Lo cual implica un nuevo proceso aleatorio.

```
# 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 -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%  85%  90%  95% 
## 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 6.95 7.30 7.65 8
```

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 MISSING 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
```

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.

[8] Es probable que a usted le salga distintos números ya que el algoritmo de selección depende de la versión del software. Este libro se desarrolló usando la versión `4.0.3` de R.

```r
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:

```r
# 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.

```r
# 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`

```r
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)
```

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

[9]Esto no implica que nos salga los mismos resultados que anteriormente, por más que se ha colocado la misma semilla, los argumentos de la sintaxis anterior han variado. Lo cual implica un nuevo proceso aleatorio.

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, por último, se mostrará como seleccionar uno o varios elementos de un vector.

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.

```
# Escogiendo aleatoriamente 5 elementos.

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

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

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] 4 1 6 9 2
```

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)
```

```
##  [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:

```r
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.

[10] Es probable que a usted le salga distintos números ya que el algoritmo de selección depende de la versión del software. Este libro se desarrolló usando la versión `4.0.3` de R.

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"  "MUJER"  "HOM
##  [12] "MUJER"  "MUJER"  "MUJER"  "HOMBRE" "HOMBRE" "MUJER"  "MUJER"  "MUJER"  "MUJER"  "MUJER"  "MUJ
##  [23] "MUJER"  "HOMBRE" "HOMBRE" "MUJER"  "HOMBRE" "HOMBRE" "MUJER"  "MUJER"  "MUJER"  "HOMBRE" "HOM
##  [34] "HOMBRE" "HOMBRE" "MUJER"  "MUJER"  "MUJER"  "MUJER"  "HOMBRE" "MUJER"  "MUJER"  "HOMBRE" "MUJ
##  [45] "MUJER"  "MUJER"  "MUJER"  "MUJER"  "MUJER"  "HOMBRE" "HOMBRE" "HOMBRE" "MUJER"  "HOMBRE" "MUJ
##  [56] "MUJER"  "HOMBRE" "HOMBRE" "HOMBRE" "HOMBRE" "HOMBRE" "HOMBRE" "MUJER"  "MUJER"  "HOMBRE" "MUJ
##  [67] "MUJER"  "HOMBRE" "MUJER"  "MUJER"  "MUJER"  "MUJER"  "HOMBRE" "HOMBRE" "MUJER"  "HOMBRE" "MUJ
##  [78] "MUJER"  "MUJER"  "MUJER"  "MUJER"  "HOMBRE" "HOMBRE" "HOMBRE" "HOMBRE" "HOMBRE" "MUJER"  "MUJ
##  [89] "HOMBRE" "MUJER"  "MUJER"  "MUJER"  "HOMBRE" "HOMBRE" "HOMBRE" "MUJER"  "MUJER"  "HOMBRE" "MUJ
## [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`.

```
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"  "MUJER"  "HOM
##  [12] "MUJER"  "MUJER"  "MUJER"  "HOMBRE" "MUJER"  "HOMBRE" "MUJER"  "MUJER"  "MUJER"  "MUJER"  "MUJ
##  [23] "MUJER"  "MUJER"  "MUJER"  "MUJER"  "MUJER"  "MUJER"  "HOMBRE" "MUJER"  "MUJER"  "MUJER"  "MUJ
##  [34] "MUJER"  "HOMBRE" "MUJER"  "HOMBRE" "HOMBRE" "MUJER"  "MUJER"  "HOMBRE" "MUJER"  "HOMBRE" "MUJ
##  [45] "MUJER"  "HOMBRE" "MUJER"  "HOMBRE" "HOMBRE" "HOMBRE" "HOMBRE" "MUJER"  "MUJER"  "MUJER"  "HOM
##  [56] "HOMBRE" "HOMBRE" "MUJER"  "MUJER"  "MUJER"  "MUJER"  "MUJER"  "MUJER"  "HOMBRE" "MUJER"  "MUJ
##  [67] "MUJER"  "MUJER"  "MUJER"  "MUJER"  "MUJER"  "HOMBRE" "MUJER"  "MUJER"  "MUJER"  "MUJER"  "HOM
##  [78] "MUJER"  "MUJER"  "MUJER"  "MUJER"  "HOMBRE" "MUJER"  "HOMBRE" "HOMBRE" "MUJER"  "MUJER"  "HOM
##  [89] "MUJER"  "MUJER"  "MUJER"  "HOMBRE" "MUJER"  "MUJER"  "HOMBRE" "HOMBRE" "MUJER"  "MUJER"  "MUJ
## [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.

```
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.

```
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.

[11] Esto no implica que nos salga los mismos resultados que anteriormente, por más que se ha colocado la misma semilla, los argumentos de la sintaxis anterior han variado. Lo cual implica un nuevo proceso aleatorio.

```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 SELECCIÓN DE ELEMENTOS.