Estructura de datos en R

Acezurita

19/6/2022

Vector

Es un arreglo de un solo tipo de dato, no se puede mezclar
R tiene varios tipos de datos.
1- logical ← Booleano
2- integer ← Enteros Z
3- numeric ← Reales, de tipo float por asi decirlo R
4- complex ← números complejos C
5- character ← tipo Char
Gana el de mayor jerarquía.
Del 1 al 5, va de menor a mayor jerarquía de datos, siendo 5 el mayor,esto se aplica cuando intentamos crear vectores mezclados de varios tipos de datos, por lo que R como tal va a convertir todos los demás datos en el tipo de mayor Jerarquía de un vector dado.
`c( )` : Define un Vector, la *c* significa concatenar, por ende puede unir vectores igual
`scan( )` : lee en consola valores para definir un vector
`fix( x )` : para modificar visualmente con un editor externo un vector
`rep( a, n)`: para definir un vector constante que tiene el dato a, repetido una n cantidad de veces.

Progresiones y Secuencias

Sucesión de números con el mismo intervalo, Su formula es

$$a_n = a_1 + (n-1) \cdot d$$

Donde los valores son:
a_n Es el resultado de un número en la sucesión, por ejemplo tenemos una d diferencia de 3, y si empezamos la secuencia con el número 1 a_1 tendremos que en el lugar número 5 n estará el número 13 a_n

Si deseamos saber la diferencia d de una sucesión de números, tenemos que la formula es:

$$d = \frac{(b-a)}{n-1}$$

Formulas en R
`seq(a,b,by=d)` donde a es inicio de la secuencia, b es el final de la misma y d es el intervalo de la sucesión.

Tenemos que esa formula despliega la siguiente secuencia 5, 10, 15, 20, 25, 30, 35, 40, 45, 50, 55, 60

Igual puede ser en forma decreciente:

```
seq(100,5,by=-9)
```

[1] 100 91 82 73 64 55 46 37 28 19 10

```r
seq(3,57,length.out=4)
```

[1] 3 21 39 57

```r
seq(2,length.out = 12, by=2)
```

[1] 2 4 6 8 10 12 14 16 18 20 22 24

```r
1:12
```

[1] 1 2 3 4 5 6 7 8 9 10 11 12

```r
20:-3
```

[1] 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 -1 -2 -3

```r
-(2:5)
```

[1] -2 -3 -4 -5

Concatenar Vectores

Para concatenar varios vectores se usa la función normal `c( )`
Por ejemplo para `c(rep(pi,5),1:5,-7)` daría: 3.14, 3.14, 3.14, 3.14, 3.14, 1, 2, 3, 4, 5, -7

Funciones y Ordenes de Vectores

¿Te acuerdas que debes usar un for para recorrer un array, para que tu puedas multiplicar cada elemento por algo o no se tu? Pues acá vale pito xd. Es decir, solo basta que coloques la variable(donde guardaste arreglo-VECTOR)y basta con que lo sumes, multipliques por algo y PUM! cada elemento será afectado por la constante de suma o multiplicacion,raiz, exponencial, etc !! xd

```r
x <- 1:10
x
```

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

```r
x*pi
```

```
##  [1]  3.141593  6.283185  9.424778 12.566371 15.707963 18.849556 21.991149
##  [8] 25.132741 28.274334 31.415927
```

```r
2^x
```

```
##  [1]    2    4    8   16   32   64  128  256  512 1024
```

```
sqrt(x)
```

```
##  [1] 1.000000 1.414214 1.732051 2.000000 2.236068 2.449490 2.645751 2.828427
##  [9] 3.000000 3.162278
```

También es posible usar la función `sapply(arreglo,FUN=function(arg){funcionAaplicar(arg)})` que te permite aplicar alguna función que por alguna razón no pueda aplicarse facilmente como en los ejemplos anteriores,y así de esta manera se logré sin ningún incoveniente. También de forma más directa sería poner la funcion sin el parentesis. Es decir: `sapply(arreglo,FUN=funcionAaplicar)`. Tal como se ve en el siguiente ejemplo.

```
x<-1:10
mipi <- function(ele){ele=ele+1}
sapply(x,FUN=function(ele){mipi(ele)})
```

```
##  [1]  2  3  4  5  6  7  8  9 10 11
```

```
sapply(x,FUN=mipi)
```

```
##  [1]  2  3  4  5  6  7  8  9 10 11
```

Se puede sumar, restar o multiplicar vectores.
Deben ser del mismo tamaño los vectores
Logica: Elemento 1 más elemento 1 del otro vector, segundo con el otro segundo y así sucecivamente. Así funciona las operaciones entre vectores.

```
vec1 = 1:10
vec2 = 1:10
tot = vec1+vec2
tot
```

```
##  [1]  2  4  6  8 10 12 14 16 18 20
```

Funciones de vectores 1.-`length(x)`: Calcula la longitud del vector x
2.-`max(x)`: Calcula el máximo del vector x
3.-`min(x)`: Calcula el mínimo del vector x
4.-`sum(x)`: Calcula la suma de las entradas del vector x
5.-`prod(x)`: Calcula el producto de las entradas del vector x
6.-`mean(x)`: Calcula la media aritmetica de las entradas del vector x
7.-`diff(x)`: Calcula el vector formado por las diferencias sucesivas entre entradas del vector original x
8.-`cumsum(x)`: Crea un vector con elementos que resultan de la suma de su anterior, es decir, 1+2 da 3 , luego 2+3 da 5 y así se va formando el otro vector.
8.-`sort(x)`: Ordena el vector en orden natural de los objetos que lo forman: el orden numérico creciente, orden alfabético...
9.-`rev(x)`: invierte el orden de los elementos del vector x

Sub Vectores

`vector[i]:`: Da la i-ésima entrada del vector
`vector[length(vector)]`: Al usar el lenght, nos da el total de elmentos del vector y al usar el normal del indice, nos dará el último valor del vector. Es una combinación de dos funciones.
`vector[a:b]`: Nos da un rango(subvector) de un valor inicial a a un valor final b de los valores originales del vector principal.
`vector[b:a]`: Nos da un rango(subvector) en orden inverso al especificado
`vector[-i]`: Si la i es un número, se creará un subvector con los valores originales del vector principal menos ese número en cuestión. Ahora si fuese la i un vector como tal, se extra ese vector y se transforma en un subvector.
`vector[-x]`: Si x es un vector, entonces este es su complementario.
Igual se pueden utilizar operadores lógicos:!=,!(NO),&(Y),|(O),etc. . .
`vector[seq(2,length(vector), by = 4)]`: Te permite obtener una secuencia de intevalo definido con los valores del vector original.
`vector[-seq(2,length(vector), by = 4)]`: Al llevar el signo menos, te crea un subvector con los valores del vector orignal sin los valores de la secuencia que se solicito.
`vector[c(1,3,5)]`: Genera un subvector con las posiciones solicitadas del vector original.
`vector[x>20]`: Genera un subvector con los valores mayores a dicho número del vector original.
`vector[x<10 | x>40]`: Se puede usar operadores logicos para obtener un resultado en especifico.
`vector1[vector2>0]`: Se puede usar otro vector del mismo tamaño del original para las condicionales. Y se evaluan por lugares, por ejemplo si el lugar uno es mayor a cero de vector 2, entonces se tomar el valor uno del vector uno para formar el subvector, los lugares inferiores a cero del vector 2 harán que no se tome los lugares del vector1 para formar el subvector.

```
x =c(1,3,5,8,7,9,6)
y =c(-1,-4,5,9,-4,9,7)
x[y>0]
```

```
## [1] 5 8 9 6
```

`x[2:5]= x[2:5]+3` Mira no más esta RIATA, o sea, que pedo? Es una chiogonada, puedes afectar solo una seccion del vector, aplicar lo que quieras y guardarlo ya modificado. (En este caso solo afecta a los lugares del vector sumandole un 3 a cada uno)
`x[length(x)+5]= 9`: Se añaden cinco nuevos lugares, y como se le indico el nueve, este número ocupara la nueva última posición y se rellenará con NA las demás.

Condicionales

`which(x cumple condición)`: Devuelve la posición/lugar[indice] del vector que cumple la condicion dada. Ejemplo, si pides numeros que superen el diez, te devolverá indices como 3,5,6, los cuales son los lugares de los números del vector que cumplen esa condicion.
`which.min(x)`: nos da el indice del primer elemento que sea el de valor minimo del vector. Ejemplo: Si el valor minimo es dos, y se repite varias veces, solo arroja la posición del primero.
`which(x==min(x))`: Arroja todas las pociciones del valor minimo.
`which.max(x)`: Nos da la primera posición del valor máximo.
`which(x==max(x))`: da todas las posiciones del valor máximo.
`x = Null`: El valor null es cuando no se define una variable/función, y se descartan al querer usarla con otros elementos.

NOT AVAILABLE! NA

`sum(x, na.rm= TRUE)`El parametro na.rm permite que una función no tome en cuenta las NA que tenga un vector, para que con ello logre ejecutarse correctamente, de lo contrario puede dar un rexultado NA, o en su caso, ejecutar los primeros resultados hasta toparse con una posición NA. Varias funciones traen ese parametro.
`is.na(x)`: Te muestra todas las posiciones del vector, y te dice TRUE o FALSE en caso de haber un valor NA, si lo hay, da TRUE.
`which(is.na(x))`: Solo de esta forma el which funciona correctamente con los valores NA, de lo contrario te arroja un integer(0) indicando que no se puede evaluar.
`x[which(is.na(x))]`: Te imprime el valores de las posiciones que este caso es NA xd.
`cumsum(x[!is.na(x)])`:Algunas funciones no tiene el parametro na.rm, por tal motivo se puede usar esta sintaxis para que funciones, la magia recae en el ! que hace que la función devuelva los que no son NA.
`cumsum(x[na.omit(x)])`: El na.omit(x) devuelve las posiciones del vector que no contengan el NA, no lo modifica de raiz.

Factores

Se crean a partir de un vector y estos establecen un orden en los mismos, dando lugar a niveles para ubicar las categorias presentes en el vector original, además de ordernarlo alfabeticamente si fuese el caso de letras, por ejemplo `vector1 = c("Adrian","Melisa","Eduard","Adrian","Melisa","Adrian")`.
Convirtiendolo a factor nos da:

```
[1] Adrian Melisa Eduard Adrian Melisa Adrian
Levels: Adrian Eduard Melisa
```

Se pueden usar etiquetas(labels) para renombrar los niveles, esto no solo afecta los niveles en sí, sino que renombra los valores del factor con sus respectivos labels.

```
 [1] "H" "M" "H" "H" "M" "H" "M" "M" "M" "H"
[1] "Mujer"        "Hombre"       "Hermafrodita"
 [1] Masculino Femenino  Masculino Masculino Femenino  Masculino Femenino
 [8] Femenino  Femenino  Masculino
Levels: Femenino Masculino Híbrido
```

Igualmente si tienes un vector de numeros `notas=c(1,2,3,2,4,3,4,2,1,4,3,3,2,4,1)`, se puede convertir a un factor y renombrar los niveles en palabras, pero al ordenarse los niveles, no lo hará alfabeticamente, sino que heredará la jerarquía de los números. De igual forma permite reagrupar varios niveles en categorías más compactas Por ejemplo:

```
 [1] 1 2 3 2 4 3 4 2 1 4 3 3 2 4 1
Levels: 1 2 3 4
 [1] reprobado  suficiente notable    suficiente excelente  notable
 [7] excelente  suficiente reprobado  excelente  notable    notable
[13] suficiente excelente  reprobado
Levels: reprobado suficiente notable excelente
 [1] reprobado aprobado  aprobado  aprobado  aprobado  aprobado  aprobado
 [8] aprobado  reprobado aprobado  aprobado  aprobado  aprobado  aprobado
[15] reprobado
Levels: reprobado aprobado
```

Igual puedes utilizar la funcion `ordered(vector,levels=..., labels = ... )` para que se cree un factor donde se establece una jearquia entre los diferentes niveles y esto se remarca bastante. Por ejemplo:

```
 [1] medio  alto   bajo   alto   bajo   medio  bajo   medio  alto   GOOD !
[11] alto   GOOD ! bajo
Levels: bajo < medio < alto < GOOD !
```

Listas

Hola, es una coleccion de datos del tipo Hash-map, como un tipo diccionario. Se crea unas pseudo nombre de variables para los diferentes vectores, datos que agregemos a una lista `list(nombre1 = vector1, nombre2 = variable, nombre3 = vector3,etc)`. Y además para accederlos se usa doble corchete, OJO, no se refiere a acceder a un elmento de un vector, sino para acceder al vector como tal `nombreLista[[número]]`, si se usa un solo corchete, te devuelve el vector como una lista, y por ende, pierde las propiedes de operacion de un vector. También puedes usar el nombre del vector/dato para acceder al mismo, usando el simbolo de moneda `nombreLista&nombreDelDAtoVector`

```
$name1
[1] 1 3 2 5 3 1 4 6 2

$name2
[1] "mipi" "chon" "es"   "muy"

$name3
[1] 2.45

[1] 1 3 2 5 3 1 4 6 2
[1] 2.45
```

Existen funciones basicas para las listas:
* Funcion para obtener los nombres de las etiquetas de los vectores$names(nombreLista)$ =name1, name2, name3
* Funcion para obtener toda la informacion de lista, como el toString de java $str(nombreLista)$ =

Matrices

Función `matrix( vector , nrow = filas, ncol = columnas, byrow = T/F)` te crea una matriz a partir de un vector, por ende siempre manejará el mismo tipo de variables.
1. **nrow** es para definir la cantidad de filas y simpre se usa en conjunto con el vector dado, **importante**: Debe ser multiplo del vector la matriz a genera, de lo contrario los campos vacios se rellenaran volviendo a usar los valores del vector repetidos.
2. **ncol** casi no se usa, solo en caso de dar un solo valor en vez de un vector. Por ende se especifica la cantidad de filas y columnas que se llenarán con ese único valor.
3. **byrow** por default es False, por ende, al crear la matriz, inserta los valores por filas, y al ser True, lo hace por columnas, te recomiendo crear una para que entiendas, ok? Pero lo haces.
Ejemplo de matriz:

$$\begin{pmatrix} 1 & 4 & 7 & 10 \\ 2 & 5 & 8 & 11 \\ 3 & 6 & 9 & 12 \end{pmatrix}$$

En la matriz anterior se insertaron los valores por filas, en este será por columnas con el byrow en True.
`matrix( vector, nrow = 3, byrow = T)`=

```
     [,1] [,2] [,3] [,4]
[1,]    1    2    3    4
```

```
[2,]    5    6    7    8
[3,]    9   10   11   12
```

Funciones que permiten crear matrices a partir de vectores de la misma longitud.

```
cbind(c(1,2,3),c(-1,-2,-3))
```

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

```
rbind(c(1,2,3),c(-1,-2,-3))
```

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

```
diag(c(1,2,3,4))
```

```
     [,1] [,2] [,3] [,4]
[1,]    1    0    0    0
[2,]    0    2    0    0
[3,]    0    0    3    0
[4,]    0    0    0    4
```

```
diag(5,nrow=3)
```

```
     [,1] [,2] [,3]
[1,]    5    0    0
[2,]    0    5    0
[3,]    0    0    5
```

Para acceder a un valor en la matriz, se usa `matriz[i,j]`
* **i** : es la fila
* **j** : es la columa

Si se especifica ambos, dará el valor del indice, si solo se usa uno, entonces te dará el valor de la fila/columna entera respectivamente.
Más funciones:
dig(matriz) : te devuelve los valores en diagonal de dicha matriz
nrow(matriz): devuelve la cantidad de filas
ncol(matriz): devuelve la cantidad de columnas
dim(matriz) : devuelve la cantidad de filas y columnas
sum(matriz) : suma de todos los valores de la matriz
prod(matriz): producto de todos los valores de la matriz
mean(matriz): la media de todos los valores de la matriz
colSums(matriz): devuelve la suma de las columnas **rowSums(matriz)**: devuleve la suma de las filas
colMeans(matriz): devuelve la media de las columnas **rowMeans(matriz)**: devuelve la media de las filas

Función apply()
apply(matriz , Margin =. . . , FUN = funcion): es como el sapply
* **Margin**: se coloca 1 para aplicar la funion por filas, el 2 por columnas y el c(1,2), para aplicarla a cada entrada de la matriz

```r
m3 = matrix(c(1:12),nrow=3)
cuadrado <- function(x){x^2}
apply(m3,MARGIN = c(1,2), FUN = cuadrado)
```

```
     [,1] [,2] [,3] [,4]
[1,]    1   16   49  100
[2,]    4   25   64  121
[3,]    9   36   81  144
```

Operaciones de matrices
t(matriz): Da su transpuesta, invierte columnas por filas
$+$: Suma las matrices
$*$: producto escalar por una matriz
___%*%___ : para multiplicar matrices, forma convencional
mtx.exp(matriz,n): para elevar la matriz a n, del paquete **Biodem**, no calcula las potencias exactas, las aproxima
%% : para elevar matrices, del paquete expm, no calcula las potencias exactas.
det(matriz): calcula el determinante de una matriz cuadrada
qr(matriz)$rank: calcula el rango de la matriz
solve(matriz): calcula la inversa de una matriz invertible.
solve(matriz,b): También sirve para resolver sistemas de ecuaciones lineales, para ello se introduce la matriz y b es el vector de término independientes.