Funciones y orden de vectores

Téllez Gerardo Rubén

13/3/2021

Funciones

Cuando queremos aplicar una función a cada uno de los elementos de un vector de datos, la función **sapply** nos ahorra tener que programar bucles en R.

- **sapply(nombre_del_vector, FUN=nombre_de_función):** para aplicar una función a todos los elementos del vector
- sqrt(x): calcula un nuevo vector con las raices cuadradas de cada uno de los elementos del vector x
- podemos operar un vector como si fuera una variable individual cuando sea una operación predefinida

```
x <- 1:10
q = x + pi
q
```

```
##  [1]  4.141593  5.141593  6.141593  7.141593  8.141593  9.141593 10.141593
##  [8] 11.141593 12.141593 13.141593
```

```
w <- sqrt(x)
w
```

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

```
e <- 2^x
e
```

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

```
m <- (1:10)^(1:10)
m
```

```
##  [1]           1           4          27         256        3125       46656
##  [7]      823543    16777216   387420489 10000000000
```

Funciones predefinidas para vectores

- **length(x):** calcula la longitud del vector x
- **max(x):** regresa el máximo del vector x
- **min(x):** regresa el mínimo del vector **x**
- **sum(x):** calcula la suma de las entradas del vector (cummin/ cumprod/ cummax/)
- **prod(x):** calcula el producto de las entradas del vector x
- **mean(x):** calcula la media aritmética $\mu/\bar{x}$ de las entradas del vector x
- **diff(x):** calcula y devuelve un vector con las restas sucesivas de cada entrada (el primero restado al segundo, el segundo restado al tercero, etc.)
- **cumsum:** calcula y devuelve el vector formado por las sumas acumuladas:
  - Permite definir sucesiones descritas mediante sumatorios
  - Cada entrada de **cumsum()** es la suma de entradas de x hasta su posición
- **sort(x):** ordena el vector en el orden creciente o alfabético (natural)
- **rev(x):** invierte el orden de los elementos del vector x

```
length(1:10)
```

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

```
max(1:10)
```

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

```
min(1:10)
```

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

```
sum(1:10)
```

```
## [1] 55
```

```
prod(1:10)
```

```
## [1] 3628800
```

```
sort(10:1)
```

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

```
rev(1:10)
```

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

```
f <- seq(2, 206, by=12)
a = mean(f)
paste(sprintf("Media aritmética: %f", a))
```

```
## [1] "Media aritmética: 104.000000"
```

```
b = diff(f)
b
```

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

```
c = cumsum(f)
c
```

```
##  [1]    2   16   42   80  130  192  266  352  450  560  682  816  962 1120 1290
## [16] 1472 1666 1872
```

Funciones definidas por el usuario

Defino la función:

$$\frac{n \cdot 2 + 1}{2} \cdot 1.987207$$

```
cv <- function(n){
  ((n*2 + 1)/2) * 1.987207
}
```

Un vector operado por cv con la función sapply:

```
n <- 1:3
r <- sapply(n, FUN = cv)
r
```

```
## [1] 2.980811 4.968018 6.955224
```

Coeficiente de determinación, ejemplo

En la regresión lineal tenemos el coeficiente de determinación. Cuando hago la regresión lineal directamente de una serie de puntos, necesito ver si se ajustan a una recta

```
cd = function(x){ summary(lm((1:4)~c(1:3,x)))$r.squared}
x <- 1:10
t <- sapply(x, FUN = cd)
```

```
## Warning in summary.lm(lm((1:4) ~ c(1:3, x))): essentially perfect fit: summary
## may be unreliable
```

```
t
```

```
##  [1] 0.01818182 0.40000000 0.89090909 1.00000000 0.96571429 0.91428571
##  [7] 0.86987952 0.83448276 0.80645161 0.78400000
```

Para tener los 100 primeros elementos de la fórmula:

$$5x^2 + 10x + 15$$

```
seg_grad <- function(x){(5*x^2) + (10*x) +15}
y <- sapply(0:100, FUN=seg_grad)
y
```

```
##   [1]    15    30    55    90   135   190   255   330   415   510   615   730
##  [13]   855   990  1135  1290  1455  1630  1815  2010  2215  2430  2655  2890
##  [25]  3135  3390  3655  3930  4215  4510  4815  5130  5455  5790  6135  6490
##  [37]  6855  7230  7615  8010  8415  8830  9255  9690 10135 10590 11055 11530
##  [49] 12015 12510 13015 13530 14055 14590 15135 15690 16255 16830 17415 18010
##  [61] 18615 19230 19855 20490 21135 21790 22455 23130 23815 24510 25215 25930
##  [73] 26655 27390 28135 28890 29655 30430 31215 32010 32815 33630 34455 35290
##  [85] 36135 36990 37855 38730 39615 40510 41415 42330 43255 44190 45135 46090
##  [97] 47055 48030 49015 50010 51015
```

Con la fórmula del pH:

$$pH = -log[H_3O^+]$$

```
ph <- function(M){
  -log10(M)
}

M <- seq(0.0001, 0.01, by=0.0001)

u <- sapply(M, ph)
u
```

```
##   [1] 4.000000 3.698970 3.522879 3.397940 3.301030 3.221849 3.154902 3.096910
##   [9] 3.045757 3.000000 2.958607 2.920819 2.886057 2.853872 2.823909 2.795880
##  [17] 2.769551 2.744727 2.721246 2.698970 2.677781 2.657577 2.638272 2.619789
##  [25] 2.602060 2.585027 2.568636 2.552842 2.537602 2.522879 2.508638 2.494850
##  [33] 2.481486 2.468521 2.455932 2.443697 2.431798 2.420216 2.408935 2.397940
##  [41] 2.387216 2.376751 2.366532 2.356547 2.346787 2.337242 2.327902 2.318759
##  [49] 2.309804 2.301030 2.292430 2.283997 2.275724 2.267606 2.259637 2.251812
##  [57] 2.244125 2.236572 2.229148 2.221849 2.214670 2.207608 2.200659 2.193820
##  [65] 2.187087 2.180456 2.173925 2.167491 2.161151 2.154902 2.148742 2.142668
##  [73] 2.136677 2.130768 2.124939 2.119186 2.113509 2.107905 2.102373 2.096910
##  [81] 2.091515 2.086186 2.080922 2.075721 2.070581 2.065502 2.060481 2.055517
##  [89] 2.050610 2.045757 2.040959 2.036212 2.031517 2.026872 2.022276 2.017729
##  [97] 2.013228 2.008774 2.004365 2.000000
```

Cálculo de un limite:

$$\frac{n^2}{(n^2+1)}$$

```
lim = function(n){(n^2)/(n^2 + 1)}
n = 1:100
v = sapply(n, lim)
v
```

```
##   [1] 0.5000000 0.8000000 0.9000000 0.9411765 0.9615385 0.9729730 0.9800000
##   [8] 0.9846154 0.9878049 0.9900990 0.9918033 0.9931034 0.9941176 0.9949239
##  [15] 0.9955752 0.9961089 0.9965517 0.9969231 0.9972376 0.9975062 0.9977376
##  [22] 0.9979381 0.9981132 0.9982669 0.9984026 0.9985229 0.9986301 0.9987261
##  [29] 0.9988124 0.9988901 0.9989605 0.9990244 0.9990826 0.9991357 0.9991843
##  [36] 0.9992290 0.9992701 0.9993080 0.9993430 0.9993754 0.9994055 0.9994334
##  [43] 0.9994595 0.9994837 0.9995064 0.9995276 0.9995475 0.9995662 0.9995837
##  [50] 0.9996002 0.9996157 0.9996303 0.9996441 0.9996572 0.9996695 0.9996812
##  [57] 0.9996923 0.9997028 0.9997128 0.9997223 0.9997313 0.9997399 0.9997481
##  [64] 0.9997559 0.9997634 0.9997705 0.9997773 0.9997838 0.9997900 0.9997960
##  [71] 0.9998017 0.9998071 0.9998124 0.9998174 0.9998223 0.9998269 0.9998314
##  [78] 0.9998357 0.9998398 0.9998438 0.9998476 0.9998513 0.9998549 0.9998583
##  [85] 0.9998616 0.9998648 0.9998679 0.9998709 0.9998738 0.9998766 0.9998793
##  [92] 0.9998819 0.9998844 0.9998868 0.9998892 0.9998915 0.9998937 0.9998959
##  [99] 0.9998980 0.9999000
```

Ejercicio

*Combinar las dos funciones, sort y rev, para crear una función que dado un vector x os lo devuelva ordenado en orden decreciente

```
x <- c(5, 84, 52, 98, 0, 10, 12)
ej <- function(x){
  rev(sort(x))
}
ej(x)
```

```
## [1] 98 84 52 12 10  5  0
```

De revertir el orden de las funciones sólo obtendría el vector en el orden natural

Hay un parámetro en sort(), ___decreasing = BOOL, por defecto en FALSE, que permite el mismo resultado:

```
x <- c(5, 84, 52, 98, 0, 10, 12)
sort(x, decreasing = TRUE)
```

```
## [1] 98 84 52 12 10  5  0
```