Nota 3 - Funciones y Manipulación de Datos

Santiago Casanova y Ernesto Barrios

Funciones

En esta sección vamos a aprender a definir y usar funciones para facilitar la organización y el uso de código en el futuro.

Definición de funciones

Las funciones se definen como cualquier objeto en `R`. El nombre del objeto seguido del operador de asignación `<-` pero en este caso después viene la palabra calve `function()`.

Veámoslo en acción para entender mejor este concepto.

```
divide_dos_num <- function(num1, num2){ #El código a correr por la función se escribe entre llaves
  num1/num2
}
```

Nótese que, similar a cómo no es necesario utilizar `print()` para que la consola responda, la respuesta de la función no requiere ni `print()` ni `return`.

Si llamamos a nuestra función:

```
divide_dos_num(10,5)
```

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

podemos verificar que funciona como esperamos.

Ejecución Controlada

if, if else, else e ifelse

Es posible que queramos controlar o limitar cuándo corre nestra función y cuándo no. En nuestro ejemplo no queremos que el segundo número sea cero porque obtendremos `NaN` por respuesta. Utilizaremos la función `if()` para controlar la ejecución.

```
divide_dos_num <- function(num1, num2){
  if(num2 == 0){
    resp <- "No es posible dividir entre cero"
  } else{
    resp <- num1/num2
  }
  resp
}

divide_dos_num(10,5)
```

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

```r
divide_dos_num(10,0)
```

```
## [1] "No es posible dividir entre cero"
```

Si queremos evaluar más de dos conidiones utilizamos `if` en combinación con `else if` y `else` para manejar los casos que no cumplan ninguna de las dos. Es importante notar que el orden de las condiciones si se toma en cuenta. Se evaluará la primera que se cumpla

Veamos un ejemplo:

```r
divide_dos_num <- function(num1, num2){
  if(num2 == 0){
    resp <- "No es posible dividir entre cero"
  } else if(num1 == 0){
    resp <- 0
  } else{
    resp <- num1/num2
  }
  resp
}

divide_dos_num(10,5)
```

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

```r
divide_dos_num(0,0)
```

```
## [1] "No es posible dividir entre cero"
```

```r
divide_dos_num(0,10)
```

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

Aunque nuestra segunda llamada a la función cumplía ambas condiciones, solamente obtenemos el resultado de que se cumpla la primera condicion.

Si queremos evaluar más de una condición en la misma expresión, podemos utilizar los operadores `&` y `|` que simbolizan `AND` y `OR` respectivamente.

Aplicamos esto a nuestro ejemplo anterior.

```r
divide_dos_num <- function(num1, num2){
  if(num2 == 0){
    resp <- "No es posible dividir entre cero"
  } else if(num1 == 0 | num2 == 1){ #operador OR |
    resp <- 'Es trivial'
  } else{
    resp <- num1/num2
  }
  resp
}

divide_dos_num(0,5)
```

```
## [1] "Es trivial"
```

```r
divide_dos_num(0,0)
```

```
## [1] "No es posible dividir entre cero"
```

```
divide_dos_num(20,1)
```

```
## [1] "Es trivial"
```

Otra forma de obtener un valor apartir de una expresión lógica en `R` es con la función `ifelse`. Esta condensa las funcionalidades de `if` y `else` en una función llamable con 3 parámetros. El primer parámetro es la expresión a evaluar, el segundo el resultado en caso de que la exresión sea `TRUE` y el tercero el resultado en caso de que la expresión sea `FALSE`.

Esta función es especialmente útil para darle valor a una variable u obtener una respuesta rápida a partir de una condición.

```
pos_neg <- function(numero){
  ifelse(numero > 0, 'positivo', 'negativo')
        #expresion    #resultado T #resultado F
}

pos_neg(-2)
```

```
## [1] "negativo"
```

```
pos_neg(3)
```

```
## [1] "positivo"
```

Como se podrán haber dado cuenta, en este ejemplo no estamos evaluando todas las condiciones posibles. Si corremos

```
pos_neg(0)
```

```
## [1] "negativo"
```

nos regresa como resultado que cero es negativo ya que no cumple la condición `x>0`. Para estos casos podemos anidar nuestros `ifelse` para que manejen mas de una condición.

```
pos_neg_cero <- function(numero){
  ifelse(numero == 0, 'cero', ifelse(numero > 0, 'positivo', 'negativo'))
}
```

En este caso evaluamos primero si el numero es igual a cero. Si no lo es, regresamos al `ifelse` que habñiamos planteado en la función anterior.

```
pos_neg_cero(-1)
```

```
## [1] "negativo"
```

```
pos_neg_cero(1231)
```

```
## [1] "positivo"
```

```
pos_neg_cero(0)
```

```
## [1] "cero"
```

for y while

En esta sección no analizaremos muy a fondo el funcionamiento o propósito de los ciclos `for` y `while` sino que veremos su sintáxis específica en `R`.

el `for` se escribe de la siguiente manera:

```r
for(i in c(1,2,3)){# el rango debe ser un vector por el cual pueda correr la i
  print(i)
}
```

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

Recordemos las maneras de escribir rangos que analizamos en la nota anterior. Ya sea con la sintáxis `inicio:fin`, con el uso de una función generadora como `seq()` o bien proporcionando un vector explícito como el rango buscado.

```r
#secuencia del 1 al 5
for(j in seq(5)){
  print(j)
}
```

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

```r
#de dos en dos del 3 al 10

for(k in seq(3,10, 2)){
  print(k)
}
```

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

```r
#rangos específicos (no necesitan ser numéricos)

for(l in c('opcion1', 'opcion2', 'opcion3')){
  print(l)
}
```

```
## [1] "opcion1"
## [1] "opcion2"
## [1] "opcion3"
```

Por otro lado el while es muy sencillo e intuituvo. Se escribe la función `while()`y como parámetro se proporciona una expresión que tenga como salida `TRUE` o `FALSE`.

Nuevamente, el código a correr se escribe entre llaves (`{}`)

Mientras la condición tenga salida `TRUE` se corre el código dentro de las llaves del while.

```r
i = 5
while(i>0){
  print(i)
  i = i-1
}
```

```
## [1] 5
## [1] 4
```

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

Nótese que en muchos casos es necesario inicializar la conidión antes de llamar a la función while.

Pseudo-aleatorios

`R` tiene muchas maneras de generar números aleatorios (o más puntualmente, pseudo-aleatorios). La forma más básica es con la fucnión `runif()`, la cual selecciona n números aleatorios en un rango dado, todos con la misma probabilidad (es decir, uniforme) de ser seleccionados.

Por default, el rango es 0 a 1 pero este puede ser modificado.

La sintaxis es: `runif(cantidad, mínimo, máximo)`

```
# 5 números del 0 al 1
runif(5)
```

```
## [1] 0.5951095 0.2875501 0.7167399 0.2337905 0.4758121
```

```
# 5 números del 3 al 8
runif(5, 3, 8)
```

```
## [1] 3.922748 5.677505 5.228100 5.597046 7.239335
```

Si quisieramos convertirla a una función discreta podemos aplicar la función `round()` alrededor de `runif()`

```
round(runif(5,3,8))
```

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

Nótese que aunque en ambos casos usamos los mismos argumentos en la función, obtubimos resultados diferentes (aparte de que los segundos están redondeados). Si quisiéramos obtener los mismos números cada vez que corremos el experimento podemos recurrir a la función `set.seed()`.

En su uso más básico, la funcion `set.seed()` recibe un parámetro numérico llamado semilla que hace que las fucniones sean repetibles. No importa cuándo o donde se corra la funcioón pseudo-aleatoria, si tienen la misma semilla antes, obtendremos el mismo resultado.

```
set.seed(13)
round(runif(5,100,112))
```

```
## [1] 109 103 105 101 112
```

Y si lo volvemos a correr:

```
set.seed(13)
round(runif(5,100,112))
```

```
## [1] 109 103 105 101 112
```

La consola nos regresa exactamente lo mismo.

Otra función muy común para simulación pseudo-aleatoria es `rbinom()`. Esta recibe 3 parámetros: el numero de ensayos, el numero de objetos y la probabilidad de éxito.

Para ilustrar esto, imaginémos que se tiene una moneda honesta, vamos a realizar 8 lanzamientos y queremos analizar el número de aguilas que obtenemos. El resultado de esta simulación con `rbinom()` se ve así:

```
# rbinom(número de lanzamientos, número de monedas, probabilidad de éxito (obtener aguila))
set.seed(205)
rbinom(8, 1, 0.5)
```

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

Vemos que en este caso obtubimos 5 aguilas en nuestro experimento.

Otra simulación, ahora más compleja, es la siguiente. Un equipo de futból, con 11 jugadores, sabe que la probabilid de que uno de ellos se lesione cada partido es de 5%. Podemos simular el número de lesiones en el calendario de 10 partidos con la función `rbinom()`

```
# rbinom(número de partidos, número de jugadores, probabilidad de lesión)
set.seed(46)
rbinom(10, 11, 0.05)
```

```
##  [1] 0 0 1 0 0 1 1 1 1 2
```

En este caso vemos que se lesionó 1 en el tercer partido, 1 en el sexto, etc. Si queremos obtener el total de lesionados podemos utilizar operaciones de vectores.

```
set.seed(46)
sum(rbinom(10, 11, 0.05))
```

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

Por último analizaremos la función `rnorm()`. Esta toma 3 argumentos: el número de experimentos, la media y la desviación estandar para generar números aleatorios con distribución normal (o también conocida como gaussiana). Veamos un ejemplo.

```
# 5 números con media cero y desviación estandar 1
rnorm(5)
```

```
## [1] -0.6224851  0.4292219  0.7208422  1.6993732  0.2467891
```

Si revisamos la media y desviación estandar de este experimento podemos confirmar nuestros argumentos iniciales (con cierto margen de error por el tamaño de la muestra).

```
vector_normal_estandar <- rnorm(300)

# media cercana a cero
mean(vector_normal_estandar)
```

```
## [1] 0.02438401
```

```
#desviación estandar cercana a 1
sd(vector_normal_estandar)
```

```
## [1] 1.057297
```

Podemos hacer lo mismo utilizando otros parámetros. Por ejemplo obtengamos 8 números con media 5 y desviación estandar 3.

```
rnorm(8,5,3)
```

```
## [1]  9.540575  1.207150 10.519307 -1.347276  6.183360  3.048465  3.984006
## [8]  2.467898
```

Como se podrán haber dado cuenta, r + el nombre de alguna distribución nos permite obtener muestras aleatorias de esa distribución con diferentes parámetros. Ejemplos más avanzados incluyen distribución exponencial `rexp()`, distribución Poisson `rpois()`, geométrica `rgeom()`, etc.

Manipulación de datos

En la nota anterior vimos una introducción a los arreglos o `data.frames`, comparadores lógicos y operaciones con vectores. Todos estos conceptos ahora nos serán útiles para aprender a manipular los datos que tenemos almacenados.

Recordemos cómo se ve el arreglo `mtcars`

```
head(mtcars)
```

```
##                   mpg cyl disp  hp drat    wt  qsec vs am gear carb
## Mazda RX4         21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
## Mazda RX4 Wag     21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
## Datsun 710        22.8   4  108  93 3.85 2.320 18.61  1  1    4    1
## Hornet 4 Drive    21.4   6  258 110 3.08 3.215 19.44  1  0    3    1
## Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2
## Valiant           18.1   6  225 105 2.76 3.460 20.22  1  0    3    1
```

Una forma de obtener columnas individuales es utilizando el operador `$` seguido del nombre de la columna. Si queremos que la consola nos regrese la columna `mpg` escribimos:

```
mtcars$mpg
```

```
##  [1] 21.0 21.0 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 17.8 16.4 17.3 15.2 10.4
## [16] 10.4 14.7 32.4 30.4 33.9 21.5 15.5 15.2 13.3 19.2 27.3 26.0 30.4 15.8 19.7
## [31] 15.0 21.4
```

Y el resultado es el vector que forma la columna `mpg`. Al ser un vector le podemos aplicar todas las tecnicas y operaciones que ta conocemos para los vectores. Por ejemplo, si quisiéramos obtener el dato en la pocisión dos escribimos:

```
mtcars$mpg[2]
```

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

Ahora vamos a crear nuestra propia columna. Para hacer esto, usamos la notación del operador `$` pero ahora con un nombre de columna que no exista. Después usamos el operador de asignación `<-` para asignarle algo a dicha columna.

```
mtcars$like <- rep(0, nrow(mtcars))
head(mtcars)
```

```
##                   mpg cyl disp  hp drat    wt  qsec vs am gear carb like
## Mazda RX4         21.0   6  160 110 3.90 2.620 16.46  0  1    4    4    0
## Mazda RX4 Wag     21.0   6  160 110 3.90 2.875 17.02  0  1    4    4    0
## Datsun 710        22.8   4  108  93 3.85 2.320 18.61  1  1    4    1    0
## Hornet 4 Drive    21.4   6  258 110 3.08 3.215 19.44  1  0    3    1    0
## Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2    0
## Valiant           18.1   6  225 105 2.76 3.460 20.22  1  0    3    1    0
```

En este caso utilizamos la función `rep()` para repetir el cero n veces donde n es el número de filas que tiene el arreglo `mtcars`. Sin embargo, `R` es un lenguaje con muchas comodidades y podemos asignar solo un cero y automáticamente lo recicla a lo largo de la columna.

```
mtcars$like <- 0
head(mtcars)
```

```
##                   mpg cyl disp  hp drat    wt  qsec vs am gear carb like
## Mazda RX4         21.0   6  160 110 3.90 2.620 16.46  0  1    4    4    0
## Mazda RX4 Wag     21.0   6  160 110 3.90 2.875 17.02  0  1    4    4    0
## Datsun 710        22.8   4  108  93 3.85 2.320 18.61  1  1    4    1    0
## Hornet 4 Drive    21.4   6  258 110 3.08 3.215 19.44  1  0    3    1    0
## Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2    0
## Valiant           18.1   6  225 105 2.76 3.460 20.22  1  0    3    1    0
```

Ahora nos gustaría cambiar algunos valores de esta columna. Para hacer esto seguimos exáctamente el mismo

proceso que usamos para modificar vectores. Seleccionamos el elemento que queremos y le asignamos un valor nuevo.

```
mtcars$like[18] <- 1
mtcars$like[12] <- 1
mtcars$like[2] <- 1
mtcars$like[28] <- 1
mtcars$like[20] <- 1
mtcars$like[21] <- 1

mtcars$like
```

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

De la misma forma, al ser un vector, podemos usar todas las técnicas y operaciones que conocemos que fincionan para vectores. Por ejemplo:

```
sum(mtcars$like)
```

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

```
max(mtcars$cyl)
```

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

La primera nos regresa la suma de la columna `like`. Es facil ver que nos regresará 6 ya que en la sección anterior le asignamos 6 `1` en diferentes pocisiones. La segunda nos regresa el valor máximo de la columna `cyl`.

Ahora vamos a analizar cómo podemos utilizar pruebas lógicas para obtener valores de un arreglo. Si corremos la expresión:

```
mtcars$cyl >=8
```

```
##  [1] FALSE FALSE FALSE FALSE  TRUE FALSE  TRUE FALSE FALSE FALSE FALSE  TRUE
## [13]  TRUE  TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE
## [25]  TRUE FALSE FALSE FALSE  TRUE FALSE  TRUE FALSE
```

vemos que nos regresa un vector booleano con valores `TRUE` o `FALSE` dependiendo si los elementos del vector escogido `mtcars$cyl` cumplen la condición. Ahora loq eu podemos hacer es pasar este vector lógico como argumento dentro de los corchetes del arreglo. Esto nos dará las filas que tengan `TRUE` en nuestra prueba lógica.

```
mtcars[mtcars$cyl >=8, ]
```

```
##                      mpg cyl  disp  hp drat    wt  qsec vs am gear carb like
## Hornet Sportabout   18.7   8 360.0 175 3.15 3.440 17.02  0  0    3    2    0
## Duster 360          14.3   8 360.0 245 3.21 3.570 15.84  0  0    3    4    0
## Merc 450SE          16.4   8 275.8 180 3.07 4.070 17.40  0  0    3    3    1
## Merc 450SL          17.3   8 275.8 180 3.07 3.730 17.60  0  0    3    3    0
## Merc 450SLC         15.2   8 275.8 180 3.07 3.780 18.00  0  0    3    3    0
## Cadillac Fleetwood  10.4   8 472.0 205 2.93 5.250 17.98  0  0    3    4    0
## Lincoln Continental 10.4   8 460.0 215 3.00 5.424 17.82  0  0    3    4    0
## Chrysler Imperial   14.7   8 440.0 230 3.23 5.345 17.42  0  0    3    4    0
## Dodge Challenger    15.5   8 318.0 150 2.76 3.520 16.87  0  0    3    2    0
## AMC Javelin         15.2   8 304.0 150 3.15 3.435 17.30  0  0    3    2    0
## Camaro Z28          13.3   8 350.0 245 3.73 3.840 15.41  0  0    3    4    0
## Pontiac Firebird    19.2   8 400.0 175 3.08 3.845 17.05  0  0    3    2    0
## Ford Pantera L      15.8   8 351.0 264 4.22 3.170 14.50  0  1    5    4    0
## Maserati Bora       15.0   8 301.0 335 3.54 3.570 14.60  0  1    5    8    0
```

Ponemos la prueba lógica seguida de una coma porque queremos obtener los renglones que cumplan esta condición, como lo vimos en la sección de arreglos de la nota anterior.

Si quisieramos que nos regrese estas filas pero sólamente una selección de columnas, podemos usar un vector con los índices (o los nombres) de las columnas deseadas después de la coma.

En estos próximos ejemplos agregaremos otra condición para limitar los reultados. Ahora buscamos todas las filas que cumplan que `cyl` sea mayor o igual a 8 y que `disp` sea mayor a 400.

```
#Un vector de indices columnas
mtcars[mtcars$cyl >=8 & mtcars$disp > 400, c(1,4,5)]
```

```
##                      mpg  hp drat
## Cadillac Fleetwood  10.4 205 2.93
## Lincoln Continental 10.4 215 3.00
## Chrysler Imperial   14.7 230 3.23
```

```
#Un rango de indices columnas
mtcars[mtcars$cyl >=8 & mtcars$disp > 400, 2:5]
```

```
##                     cyl disp  hp drat
## Cadillac Fleetwood    8  472 205 2.93
## Lincoln Continental   8  460 215 3.00
## Chrysler Imperial     8  440 230 3.23
```

```
#Un vecotr con nombres de columnas
mtcars[mtcars$cyl >=8 & mtcars$disp > 400, c('mpg','cyl', 'disp')]
```

```
##                      mpg cyl disp
## Cadillac Fleetwood  10.4   8  472
## Lincoln Continental 10.4   8  460
## Chrysler Imperial   14.7   8  440
```

Si sólo bucamos una sola columna, también se puede utilizar el operador `$` después de los corchetes para indicar que queremos que nos regrese esa columna. Nótese que aún es necesario escribir la coma.

```
mtcars[mtcars$cyl >=8 & mtcars$disp > 400,]$mpg
```

```
## [1] 10.4 10.4 14.7
```

De igual manera podemos notar que cuando seleccionamos más de una columna la consola nos regresa un arreglo, mientras que cuando sólo seleccionamos una columna (ya sea con índice, nombre o el operador `$`) la consola regresa un vector.

Esto es crucial ya que nos permite aplicar todas las operaciones y manipulaciones de vectores que ya conocemos.

Esta sintáxis no solo sirve para obtener los datos a través de la consola. Naturalmente también podemos asignar estos resultados a una nueva variable. Vamos a crear un *subset* de `mtcars` que sólo incluya las filas con `cyl` igual a 4.

```
cars_4_cyl <- mtcars[mtcars$cyl == 4, ]
head(cars_4_cyl)
```

```
##                 mpg cyl  disp hp drat    wt  qsec vs am gear carb like
## Datsun 710     22.8   4 108.0 93 3.85 2.320 18.61  1  1    4    1    0
## Merc 240D      24.4   4 146.7 62 3.69 3.190 20.00  1  0    4    2    0
## Merc 230       22.8   4 140.8 95 3.92 3.150 22.90  1  0    4    2    0
## Fiat 128       32.4   4  78.7 66 4.08 2.200 19.47  1  1    4    1    1
## Honda Civic    30.4   4  75.7 52 4.93 1.615 18.52  1  1    4    2    0
## Toyota Corolla 33.9   4  71.1 65 4.22 1.835 19.90  1  1    4    1    1
```

Ahora usemos lo que sabemos sobre crear columnas y números pseudo-aleatorios para crear una columna `tank` que indique el tamaño del tanque de gasolina de los coches.

```
set.seed(13)
cars_4_cyl$tank <- round(rnorm(nrow(cars_4_cyl), 40, 10))
cars_4_cyl
```

```
##                 mpg cyl  disp  hp drat    wt  qsec vs am gear carb like tank
## Datsun 710     22.8   4 108.0  93 3.85 2.320 18.61  1  1    4    1    0   46
## Merc 240D      24.4   4 146.7  62 3.69 3.190 20.00  1  0    4    2    0   37
## Merc 230       22.8   4 140.8  95 3.92 3.150 22.90  1  0    4    2    0   58
## Fiat 128       32.4   4  78.7  66 4.08 2.200 19.47  1  1    4    1    1   42
## Honda Civic    30.4   4  75.7  52 4.93 1.615 18.52  1  1    4    2    0   51
## Toyota Corolla 33.9   4  71.1  65 4.22 1.835 19.90  1  1    4    1    1   44
## Toyota Corona  21.5   4 120.1  97 3.70 2.465 20.01  1  0    3    1    1   52
## Fiat X1-9      27.3   4  79.0  66 4.08 1.935 18.90  1  1    4    1    0   42
## Porsche 914-2  26.0   4 120.3  91 4.43 2.140 16.70  0  1    5    2    0   36
## Lotus Europa   30.4   4  95.1 113 3.77 1.513 16.90  1  1    5    2    1   51
## Volvo 142E     21.4   4 121.0 109 4.11 2.780 18.60  1  1    4    2    0   29
```

Estamos creando la columna `tank` con números enteros (gracias a `round()`) con media 40 y desviación estandar 10 (con la función `rnorm()`). Para el cantidad de números aleatorios a generar utilizamos `nrow()` para que la función nos regrese los suficientes para todas las filas de nuestro arreglo.

Veamosel resumen nuestra nueva columna.

```
summary(cars_4_cyl$tank)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   29.00   39.50   44.00   44.36   51.00   58.00
```