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.

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`.