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 clave `function()`.

Dentro de los paréntesis escribimos los arguementos que queremos que reciba nuestra función. Después, entre llaves, se escribe la lógica a seguir de la función.

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 nuestra 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 condiciones 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 condición (ya que R evalúa de arriba hacia abajo y se detiene cuando encuentra una condición verdadera).

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 = 2){
  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"
```

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

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

Nótese también que podemos definir valores por *default* para los parámetros. En este caso declaramos que `num2` es igual a 2 por default y por lo tanto, si el usuario no proporciona un valor para ese parámetro, la función puede correr con un valor predeterminado.

```r
divide_dos_num(10) #Será dividido entre dos porque no proporcionamos otro parámetro
```

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

Otra forma de controlar la ejecución con una operación 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 expresión 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.

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

pos_neg(-2)
```

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

```r
pos_neg(3)
```

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

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

```r
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 más de una condición.

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

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

```r
pos_neg_cero(-1)
```

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

```r
pos_neg_cero(1231)
```

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

```r
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 sintaxis específica en `R`.

el `for` se escribe de la siguiente manera:

```
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 sintaxis `inicio:fin`, con el uso de una función generadora como `seq()` o bien proporcionando un vector explícito como el rango buscado.

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

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

```
#secuencia 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
```

```
#rangos específicos (no necesitan ser numéricos). Estos son útiles cuando tratamos con columnas de un d

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

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

Por otro lado el while es muy sencillo e intuitivo. 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.

```
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 condición 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 función `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] 0.50983629 0.41249484 0.07316865 0.22262843 0.23699363
```

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

```
## [1] 6.718136 5.263347 7.926920 7.054992 4.442439
```

Si quisiéramos convertirla en una función discreta podemos aplicar la función `round()` alrededor de `runif()`

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

```
## [1] 7 8 3 4 4
```

Nótese que aunque en ambos casos usamos los mismos argumentos en la función, obtuvimos 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 función `set.seed()` recibe un parámetro numérico llamado semilla que hace que las funciones sean repetibles. No importa cuándo o dónde se corra la función 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 número de ensayos, el número de objetos y la probabilidad de éxito. (Es decir una simulación binomial)

La sintaxis es: `rbinom(n, tamaño de ensayo, probabilidad)`

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

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

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

Vemos que en este caso obtuvimos 5 águilas en nuestro experimento.

Veamos una simulación más compleja. Un equipo de futból, con 11 jugadores, sabe que la probabilidad 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 estándar para generar números aleatorios con distribución normal (o también conocida como gaussiana). Veamos un ejemplo.

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

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

Si revisamos la media y desviación estándar 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 estándar 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 estándar 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.