Nota2 - Introducción

Santiago Casanova y Ernesto Barrios

Uso Básico

Ahora vamos a empezar a familiarizarnos con el ambiente de R y específicamente su sintaxis especial.

Declaración de variables

`R` tiene un operador especial para asignar valores a variables, diferente a otros lenguajes de programación. Además, las variables son flexibles y pueden pasar de contener un tipo de dato a otro sin problema. Por los mismo no es necesario especificar el tipo de dato como se hace en C o en Java. El operador de asignación es `<-`. También se puede asignar variables con `=` pero, por convención, `=` se reserva para operaciones dentro de funciones o paréntesis.

Vamos a asignar algunos valores a una serie de variables:

```
numero <- 3
numero2 <- 1234.56
entero <- 20L
texto1 <- "ejemplo"
texto2 <- 'tambien se puede con comillas simples'
booleano <- T
booleano = TRUE
booleano2 = F
booleano2 = FALSE

typeof(numero)
```

```
## [1] "double"
```

```
typeof(entero)
```

```
## [1] "integer"
```

Como describimos, no se necesita especificar que tipo de dato queremos en cada variable ya que esto puede cambiar mas adelante. Hablando de tipos de datos, vamos a ver cuáles son las opciones que maneja `R`.

Tipos de Datos

1. Numérico: En principio no hay diferencia entre `integer` y `double` o como también es conocido, `float`.
- Si se quiere declarar específicamente como integer se puede escribir con una `L` al final del número. Ejemplo: `2L`. De lo contrario `R` lo guardará como `double`.
2. Caracter: No hay diferencia entre `string` y `character`. En `R` todos los literales entre comillas son considerados modo y y tipo `character`.
3. Booleano (lógico): Como vimos en el ejemplo anterior, se escribe `TRUE` o `FALSE` todo con mayúsculas, o bién, sólo `T` o `F`.

Valores no-disponibles (Not Available)

`R` tiene varias maneras de manejar los valores no disponibles dependiendo del tipo de dato. Un valor no disponible actúa como un marcador de posición en una estructura donde *debería* haber un dato. Esto lo veremos más a fondo cuando llegeuemos al tema de estructuras de datos.

Cuando el valor no esta disponible se representa con `NA`. De la misma manera, los valores `NA` pueden ser más especificos. `NA_real_`, `NA_integer_`, `NA_character_` y `NA_complex_` describen puntualmente el tipo de dato que falta pero en ultima instancia todos son tratados como `NA` por `R`. Por ejemplo:

```
NA
```

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

```
NA_real_
```

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

tienen la misma salida.

Además del `NA`, R reconoce `NaN` como **Not a Number** y es especifico para cuando el resultado de una operación matemática resulta en algo imposible. Por ejemplo la divisón de 0 entre 0. Es diferente a `NA` porque no indica que falte un valor sino que el velor resulto en un no-numero.

```
print(0/0)
```

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

Por ultimo tenemos el valor `NULL` que indica la ausencia de todo dato. Puede ser usado para desasignar variables. Por ejemplo:

```
var <- 'Ejemplo'
print(var)
```

```
## [1] "Ejemplo"
```

```
var <- NULL
print(var)
```

```
## NULL
```

Operadores

La mayoría de los operadores son muy similares o iguales a otros lenguajes de programación por lo que no los analizaremos a fondo.

```
#suma
print(20 + 3)
```

```
## [1] 23
```

```
#resta
print(20 - 3)
```

```
## [1] 17
```

```
#multiplicación
print(20 * 3)
```

```
## [1] 60
```

```
#división
print(20 / 3)
```

```
## [1] 6.666667
```

```
#potencia (diferente a otros)
print(20 ^ 3)
```

```
## [1] 8000
```

```
#módulo
print(20 %% 3)
```

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

Operadores lógicos

En R, los operadores lógicos básicos son prácticamente iguales a otros lenguajes de programaxión pero además, R reconoce una serie de pruebas lógicas especiales que veremos a fondo mas adelante.

```
#mayor que (o mayor/igual)
print(20 > 3) # >=
```

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

```
#menor que (o menor/igual)
print(20 < 3) # <=
```

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

```
#comparar (igualdad)
20 == 3
```

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

```
#comparar (desigualdad)
20 != 3
```

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

Hasta ahora hemos usado la funcion `print()` para obtener una respuesta de la consola. Sin embargo, como podemos ver en las dos últimas salidas de los ejemplos anteriores, R también regresa el resultado de una operación sin el uso específico de `print()` cuando está en una sesión. De ahora en adelante lo omitiremos en nuestros ejemplos.

Otra forma de obtener valores lógicos es revisando el tipo de dato que tenemos en una variable. Para esto, R nos presenta una familila de funciones que revisan el tipo de dato y regresan un valor lógico. La familila de funciones `is.*` ejecuta exactamente este mismo proceso en una sola función.

```
palabra <- 'palabra'
numeros <- 123.2

is.character(palabra)
```

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

```
is.character(numeros)
```

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

```
is.numeric(palabra)
```

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

```
is.numeric(numeros)
```

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

Estructuras de Datos

Vector Atómico

La estructura más básica para recopilar datos en R es un **Vector Atómico**. Estos se construyen con la sintáxis `c(a1, a2)` que concatena los valores deseados. Un vector atómico sin valores no es considerado una estructura de datos. R lo reconoce como un valor nulo `NULL` que analizamos en la sección anterior.

```
c()
```

```
## NULL
```

Ahora vemaos un par de ejemplos de vectores.

```
c(1,2,3,4,5)
```

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

```
primer_vector <- c(1,2,3, 'a', 'b', 'c', NA, NA_real_ ,TRUE)
print(primer_vector)
```

```
## [1] "1"    "2"    "3"    "a"    "b"    "c"    NA     NA     "TRUE"
```

El primer vector concatena exclusivamente datos numéricos, por lo que el vector entero será de tipo numerico. Sin embargo, el segundo vector incluye datos de tipo numérico, caracter y lógico. Por esto, R considera a todo el vector de tipo `character` y en la respuesta los números de nuestro vector están encerrados en comillas.

Es importante notar que los `NA` no se convierten a tipo `character` porque `NA` no es un tipo de dato, representa la ausencia de uno.

Nótese que el vector atómico es la estructura *default* de R. Por lo tanto si no se especifica cómo guardar una colección de datos, R la guardará en un vector

```
#La sintáxis de dos números separados por dos puntos ":" denota un rango
vec <- 1:5
vec
```

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

Vemos que la consola nos regresa un vector.

Lista

Las listas agregan un grado de complejidad a la estructura "lineal" que son los vectores. Cada elemento de una lista puede ser una estructura de dato diferente por lo que siempre mantendrá los datos con su tipo original. Para construir una lista se utiliza la función `list()`

Las listas pueden llegar a ser muy complejas (y por lo tanto muy útiles para recopilar datos diversos) pero en su forma más básica las podemos usar como una especie de vector que respeta los tipo de dato.

```
list(1,2,3,4,5)
```

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

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

```
primer_lista <- list(1,2, 'a', 'b', NA,TRUE)
print(primer_lista)
```

```
## [[1]]
## [1] 1
##
## [[2]]
## [1] 2
##
## [[3]]
## [1] "a"
##
## [[4]]
## [1] "b"
##
## [[5]]
## [1] NA
##
## [[6]]
## [1] TRUE
```

Inmediatamente podemos ver la diferencia entre las respuestas de la consola con vectores y con listas. Todos los valores que ingresamos mantienen su tipo original (los valores numéricos no tienen comillas alrededor), pero además cada elemento está separado en un renglón.

Como dijimos, las listas no solo pueden contener datos individuales, sino también estructuras. Si combinamos los conceptos de lista y vector, podemos crear una lista de vectores, cada uno de un tipo diferente.

```
lista_compleja <- list(c(1,2,4, 12873812), c('a', 'b','c','g', 'zh', 'ya'), c(NA, NA_real_, NaN), 201)
lista_compleja
```

```
## [[1]]
## [1]        1        2        4 12873812
##
## [[2]]
## [1] "a"  "b"  "c"  "g"  "zh" "ya"
##
## [[3]]
## [1]  NA  NA NaN
##
## [[4]]
## [1] 201
```

Las listas permiten guardar datos en varios niveles y dimensiones.

Esto presenta la pregunta ¿cómo accedemos a estos datos?

Acceso a datos con indices

Los indices en `R` inician en `1` y se escriben dentro de `[]` para acceder al dato correspondiente.

En nuestro ejemplo del prmer vector:

```
#La segunda pocisión del vector
primer_vector[2]
```

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

Esto nos indica que en la pocisión 2 de `primer_vector` tenemos un "2" de tipo `character` (notar comillas).

Sin embargo, el proceso para las listas en un poco diferente. Si intentamos hacer lo mismo, obtendremos una lista que contiene al dato buscado, en lugar de el dato buscado pos si solo.

```
lista_compleja[1]
```

```
## [[1]]
## [1]        1        2        4 12873812
```

Para obtener sólamente el primer elemento de la lista se usa un corchete doble `[[]]`

```
lista_compleja[[1]]
```

```
## [1]        1        2        4 12873812
```

Ahora vemos que nuestra salida es un vector, el elemento que teníamos guardado como primer elemento de `lista_compleja`. Como nuestro reultado es un vector esto significa que va a responder exactamente como lo haría un vector. Si quisiéramos acceder al tercer elemento de este vector escribimos:

```
lista_compleja[[1]][3]
```

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

Ya que la lista simplemente esta guardando al vector como un elemento, todas las propiedades de la estructura funcionan igual cuando la llamamos a través de indices.

Data Frames

Hasta ahora hemos visto estructuras de datos simples, en el caso de los vectores, o compuestas sin relación, en el caso de las listas. Los data frames nos permiten organizar datos que se relacionan entre si. Podemos pensar que los data frames se comportan como lo haría una hoja de excel, con filas y columnas que contienen datos de cualquier tipo.

Para crear un data fram usamos la función `data.frame()` y le propocionamos vectores atómicos como columnas.

```
#Creamos un arreglo con dos vectores. Cada uno corresponde a una columna de nuestro arreglo
arreglo <- data.frame(nombre = c('Ernesto', 'Santiago'),
                      clave = c(21,18),
                      booleano = c(T,F))

arreglo
```

```
##     nombre clave booleano
## 1  Ernesto    21     TRUE
## 2 Santiago    18    FALSE
```

Nótese que, como las columnas están generadas por vectores atómicos, cada una solo puede tener un tipo de dato.

Para acceder a los valores individuales usamos la misma sintaxis que usamos para los vectores pero ahora con dos pocisiones separadas por una coma. La primera corresponde a la **fila** que queremos y la segunda a la **columna**.

```r
#Primera columna y primera fila
arreglo[1,1]
```

```
## [1] "Ernesto"
```

Es el dato correspondiente a la primera fila y primera columna. Si solo proporcionamos un índice sin usar la coma, `R` regresa la columna que corresponda al indice.

```r
#La primera columna
arreglo[1]
```

```
##     nombre
## 1  Ernesto
## 2 Santiago
```

Explorar las estructuras (??)

length dim summary

Operaciones de vectores

Cuando tenemos una serie de datos numéricos guardada en un vector le podemos aplicar varias operaciones como suma, promedio (media) o mediana, entre otras. Por ejemplo:

```r
vector_numerico <- c(123,546.7,333,32,1)

#sumar todas las entradas
sum(vector_numerico)
```

```
## [1] 1035.7
```

```r
#obtener el promedio
mean(vector_numerico)
```

```
## [1] 207.14
```

```r
#obtener la mediana
median(vector_numerico)
```

```
## [1] 123
```

```r
#redondear los valores
round(vector_numerico)
```

```
## [1] 123 547 333  32   1
```

Uso de RStudio

Hasta ahora hemos usado solamente la consola para interactuar con R pero hay maneras mas amigables de escribir comandos y recibir respuestas.

Una interfaz o IDE (Integrated Developement Enviroment) como RStudio nos permite escribir archivos ejecutables en los que podemos incluir todo un programa y editarlo sin tener que correr cada linea todas las veces.

Aquí incluimos una guía para instalar RStudio en diferentes sistemas operativos.

Windows

1. Primero visitar https://www.rstudio.com/products/rstudio/download/#download.
2. Hacer click en **Download** en la sección que dice *RStudio Desktop*.
3. Hacer click en **Download RStuio for Windows** si aparece el botón.
4. Si no aparece el botón, buscar *Windows 10/11* (o el que corresponda) en la lista **All Installers** y guardar el archivo ejecutable.
5. Correr el archivo .exe y seguir las instrucciones de instalación.

Mac

1. Primero visitar https://www.rstudio.com/products/rstudio/download/#download.
2. Hacer click en **Download** en la sección que dice *RStudio Desktop*.
3. Hacer click en **Download RStuio for Mac** si aparece el botón.
4. Si no aparece el botón, buscar *macOS 10.15+* (o el que corresponda) en la lista **All Installers** y guardar el archivo ejecutable.
5. Correr el archivo .dmg y seguir las instrucciones de instalación.