(../03-seeking-help/index.html) R para Análisis Científicos Reproducibles (../) (../05-data-struct part2/

Estructuras de datos

Hoja de ruta

Enseñando: 40 min
Prácticas: 15 min
Preguntas

- ¿Cómo puedo leer datos en R?
- ¿Cuáles son los tipos de datos básicos en R?
- ¿Cómo represento la información categórica en R?

Objetivos

- Conocer los distintos tipos de datos.
- Comenzar a explorar los *data frames* y entender cómo se relacionan con **vectors**, **factors** y **lists**.
- Ser capaz de preguntar sobre el tipo, clase y estructura de un objeto en R.
- Conocer y entender qué es coerción y cuáles son los distintos tipos de coerciones.

Palabras clave

Comando : Significado

data set : conjunto de datos

`c` : combinar

`dim` : dimensión

`nrow` : número de filas

`ncol` : número de columnas

Una de las características más poderosas de R es su habilidad para manejar datos tabulares - como los que puedes tener en una planilla de cálculo o un archivo CSV. Comencemos creando un **dataset** llamado `gatos` que se vea de la siguiente forma:

R

```
color,peso,le_gusta_cuerda
mixto,2.1,1
negro,5.0,0
atigrado,3.2,1
```

Podemos usar la función `data.frame` para crearlo.

R

```
gatos <- data.frame(color = c("mixto", "negro", "atigrado"),
                    peso = c(2.1, 5.0, 3.2),
                    le_gusta_cuerda = c(1, 0, 1))
gatos
```

Output

```
     color peso le_gusta_cuerda
1    mixto  2.1               1
2    negro  5.0               0
3 atigrado  3.2               1
```

📌 Consejo: Edición de archivos de texto en R

Alternativamente, puedes crear el archivo `data/feline-data.csv` usando un editor de texto (Nano), o en RStudio usando el ítem del Menú **File -> New File -> Text File**.

Podemos leer el archivo en R con el siguiente comando:

R

```
gatos <- read.csv(file = "data/feline-data.csv")
gatos
```

Output

```
    coat weight likes_string
1 calico    2.1            1
2  black    5.0            0
3  tabby    3.2            1
```

La función `read.table` se usa para leer datos tabulares almacenados en un archivo de texto donde las columnas de datos están separadas por caracteres de puntuación, por ejemplo archivos CSV (csv = valores separados por comas). Las tabulaciones y las comas son los caracteres de puntuación más utilizados para separar o delimitar datos en archivos csv. Para mayor comodidad, R proporciona otras 2 versiones de “read.table”. Estos son: `read.csv` para archivos donde los datos están separados por comas y `read.delim` para archivos donde los datos están separados por tabulaciones. De estas tres funciones, `read.csv` es el más utilizado. Si fuera necesario, es posible anular o modificar el delimitador predeterminado para `read.csv` y ` read.delim`.

Podemos empezar a explorar el **dataset** inmediatamente proyectando las columnas usando el operador `$` :

R

```
gatos$peso
```

Output

```
NULL
```

R

```
gatos$color
```

Output

```
NULL
```

Podemos hacer otras operaciones sobre las columnas. Por ejemplo, podemos aumentar el peso de todos los gatos con:

R

```
gatos$peso + 2
```

Output

```
numeric(0)
```

Podemos imprimir los resultados en una oración

R

```
paste("El color del gato es", gatos$color)
```

Output

```
[1] "El color del gato es "
```

Pero qué pasa con:

R

```
gatos$peso + gatos$color
```

Output

```
integer(0)
```

Si adivinaste que el último comando iba a resultar en un error porque `2.1` más `"negro"` no tiene sentido, estás en lo cierto - y ya tienes alguna intuición sobre un concepto importante en programación que se llama *tipos de datos*.

No importa cuán complicado sea nuestro análisis, todos los datos en R se interpretan con uno de estos tipos de datos básicos. Este rigor tiene algunas consecuencias importantes.

Hay 5 tipos de datos principales: `double` , `integer` , `complex` , `logical` and `character` .

Podemos preguntar cuál es la estructura de datos si usamos la función `class` :

R

```
class(gatos$color)
```

Output

```
[1] "NULL"
```

R

```
class(gatos$peso)
```

Output

```
[1] "NULL"
```

También podemos ver que `gatos` es un **data.frame** si usamos la función `class` :

R

```
class(gatos)
```

Output

```
[1] "data.frame"
```

Vectores y Coerción de Tipos

Para entender mejor este comportamiento, veamos otra de las estructuras de datos en R: el **vector**.

Un vector en R es esencialmente una lista ordenada de cosas, con la condición especial de que *todos los elementos en un vector tienen que ser del mismo tipo de datos básico*. Si no eliges un tipo de datos, por defecto R elige el tipo de datos **logical**. También puedes declarar un vector vacío de cualquier tipo que quieras.

Una indicación del número de elementos en el vector - específicamente los índices del vector, en este caso `[1:3]` y unos pocos ejemplos de los elementos del vector - en este caso **strings** vacíos.

Podemos ver que `gatos$peso` es un vector usando la funcion `str` .

R

```
str(gatos$peso)
```

Output

```
 NULL
```

Las columnas de datos que cargamos en **data.frames** de R son todas vectores y este es el motivo por el cual R requiere que todas las columnas sean del mismo tipo de datos básico.

Discusión 1

¿Por qué R es tan obstinado acerca de lo que ponemos en nuestras columnas de datos? ¿Cómo nos ayuda esto?

Discusión 1

También puedes crear vectores con contenido explícito con la función **combine** o `c()` :

R

```
mi_vector <- c(2,6,3)
mi_vector
```

Output

```
[1] 2 6 3
```

R

```
str(mi_vector)
```

Output

```
 num [1:3] 2 6 3
```

Dado lo que aprendimos hasta ahora, ¿qué crees que hace el siguiente código?

R

```
otro_vector <- c(2,6,'3')
```

Esto se denomina *coerción de tipos de datos* y es motivo de muchas sorpresas y la razón por la cual es necesario conocer los tipos de datos básicos y cómo R los interpreta. Cuando R encuentra una mezcla de tipos de datos (en este caso **numeric** y **character**) para combinarlos en un vector, va a forzarlos a ser del mismo tipo.

Considera:

R

```
vector_coercion <- c('a', TRUE)
str(vector_coercion)
```

Output

```
 chr [1:2] "a" "TRUE"
```

R

```
otro_vector_coercion <- c(0, TRUE)
str(otro_vector_coercion)
```

Output

```
 num [1:2] 0 1
```

Las reglas de coerción son: `logical` -> `integer` -> `numeric` -> `complex` -> `character` , donde -> se puede leer como *se transforma en*. Puedes intentar forzar la coerción de acuerdo a esta cadena usando las funciones `as.` :

R

```
vector_caracteres <- c('0','2','4')
vector_caracteres
```

Output

```
[1] "0" "2" "4"
```

R

```
str(vector_caracteres)
```

Output

```
 chr [1:3] "0" "2" "4"
```

R

```
caracteres_coercionados_numerico <- as.numeric(vector_caracteres)
caracteres_coercionados_numerico
```

Output

```
[1] 0 2 4
```

R

```
numerico_coercionado_logico <- as.logical(caracteres_coercionados_numerico)
numerico_coercionado_logico
```

Output

```
[1] FALSE  TRUE  TRUE
```

Como puedes notar, es sorprendete ver qué pasa cuando R forza la conversión de un tipo de datos a otro. Es decir, si tus datos no lucen como pensabas que deberían lucir, puede ser culpa de la coerción de tipos. Por lo tanto, asegúrate que todos los elementos de tus vectores y las columnas de tus **data.frames** sean del mismo tipo o te encontrarás con sorpresas desagradables.

Pero la coerción de tipos también puede ser muy útil. Por ejemplo, en los datos de `gatos`, `le_gusta_cuerda` es numérica, pero sabemos que los 1s y 0s en realidad representan **`TRUE`** y **`FALSE`** (una forma habitual de representarlos). Deberíamos usar el tipo de datos **`logical`** en este caso, que tiene dos estados: **`TRUE`** o **`FALSE`**, que es exactamente lo que nuestros datos representan. Podemos convertir esta columna al tipo de datos **`logical`** usando la función `as.logical`:

R

```
gatos$le_gusta_cuerda
```

Output

```
NULL
```

R

```
class(gatos$le_gusta_cuerda)
```

Output

```
[1] "NULL"
```

R

```
gatos$le_gusta_cuerda <- as.logical(gatos$le_gusta_cuerda)
```

Error

```
Error in `$<-.data.frame`(`*tmp*`, le_gusta_cuerda, value = logical(0)): replacement has 0 rows, data has 3
```

R

```
gatos$le_gusta_cuerda
```

Output

```
NULL
```

R

```
class(gatos$le_gusta_cuerda)
```

Output

```
[1] "NULL"
```

La función **combine**, `c()`, también agregará elementos al final de un vector existente:

R

```
ab <- c('a', 'b')
ab
```

Output

```
[1] "a" "b"
```

R

```
abc <- c(ab, 'c')
abc
```

Output

```
[1] "a" "b" "c"
```

También puedes hacer una serie de números así:

R

```
mySerie <- 1:5
mySerie
```

Output

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

R

```
str(mySerie)
```

Output

```
 int [1:5] 1 2 3 4 5
```

R

```
class(mySerie)
```

Output

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

Finalmente, puedes darle nombres a los elementos de tu vector:

R

```
names(mySerie) <- c("a", "b", "c", "d", "e")
mySerie
```

Output

```
a b c d e
1 2 3 4 5
```

R

```
str(mySerie)
```

Output

```
 Named int [1:5] 1 2 3 4 5
 - attr(*, "names")= chr [1:5] "a" "b" "c" "d" ...
```

R

```
class(mySerie)
```

Output

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

Desafío 1

Comienza construyendo un vector con los números del 1 al 26. Multiplica el vector por 2 y asigna al vector resultante, los nombres de A hasta Z (Pista: hay un vector pre-definido llamado `LETTERS`)

Solución del desafío 1

Factores

Otra estructura de datos importante se llama **factor**.

R

```
str(gatos$color)
```

Output

```
 NULL
```

Los factores usualmente parecen caracteres, pero se usan para representar información categórica. Por ejemplo, construyamos un vector de **strings** con etiquetas para las coloraciones para todos los gatos en nuestro estudio:

R

```
capas <- c('atigrado', 'tortoiseshell', 'tortoiseshell', 'negro', 'atigrado')
capas
```

Output

```
[1] "atigrado"      "tortoiseshell" "tortoiseshell" "negro"
[5] "atigrado"
```

R

```
str(capas)
```

Output

```
 chr [1:5] "atigrado" "tortoiseshell" "tortoiseshell" "negro" "atigrado"
```

Podemos convertir un vector en un **factor** de la siguiente manera:

R

```
categorias <- factor(capas)
class(categorias)
```

Output

```
[1] "factor"
```

R

```
str(categorias)
```

Output

```
 Factor w/ 3 levels "atigrado","negro",..: 1 3 3 2 1
```

Ahora R puede interpretar que hay tres posibles categorías en nuestros datos - pero también hizo algo sorprendente: en lugar de imprimir los **strings** como se las dimos, imprimió una serie de números. R ha reemplazado las categorías con índices numéricos, lo cual es necesario porque muchos cálculos estadísticos usan esa representación para datos categóricos:

R

```
class(capas)
```

Output

```
[1] "character"
```

R

```
class(categorias)
```

Output

```
[1] "factor"
```

Desafío 2

¿Hay algún **factor** en nuestro **data.frame** `gatos` ? ¿Cuál es el nombre? Intenta usar `?read.csv` para darte cuenta cómo mantener las columnas de texto como vectores de caracteres en lugar de factores; luego escribe uno o más comandos para mostrar que el **factor** en `gatos` es en realidad un vector de caracteres cuando se carga de esta manera.

Solución al desafío 2

En las funciones de modelado, es importante saber cuáles son los niveles de referencia. Se asume que es el primer factor, pero por defecto los factores están etiquetados en orden alfabetico. Puedes cambiar esto especificando los niveles:

R

```
misdatos <- c("caso", "control", "control", "caso")
factor_orden <- factor(misdatos, levels = c("control", "caso"))
str(factor_orden)
```

Output

```
 Factor w/ 2 levels "control","caso": 2 1 1 2
```

En este caso, le hemos dicho explícitamente a R que “control” debería estar representado por 1, y “caso” por 2. Esta designación puede ser muy importante para interpretar los resultados de modelos estadísticos!

Listas

Otra estructura de datos que querrás en tu bolsa de trucos es `list`. Una lista es más simple en algunos aspectos que los otros tipos, porque puedes poner cualquier cosa que tú quieras en ella:

R

```
lista <- list(1, "a", TRUE, 1+4i)
lista
```

Output

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

[[2]]
[1] "a"

[[3]]
[1] TRUE

[[4]]
[1] 1+4i
```

R

```
otra_lista <- list(title = "Numbers", numbers = 1:10, data = TRUE )
otra_lista
```

Output

```
$title
[1] "Numbers"

$numbers
 [1]  1  2  3  4  5  6  7  8  9 10

$data
[1] TRUE
```

Ahora veamos algo interesante acerca de nuestro **data.frame**; ¿Qué pasa si corremos la siguiente línea?

R

```
typeof(gatos)
```

Output

```
[1] "list"
```

Vemos que los **data.frames** parecen listas 'en su cara oculta' - esto es porque un **data.frame** es realmente una lista de vectores y factores, como debe ser - para mantener esas columnas que son una combinación de vectores y factores, el **data.frame** necesita algo más flexible que un vector para poner todas las columnas juntas en una tabla. En otras palabras, un `data.frame` es una lista especial en la que todos los vectores deben tener la misma longitud.

En nuestro ejemplo de `gatos`, tenemos una variable **integer**, una **double** y una **logical**. Como ya hemos visto, cada columna del **data.frame** es un vector.

R

```
gatos$color
```

Output

```
NULL
```

R

```
gatos[,1]
```

Output

```
[1] "calico" "black"  "tabby"
```

R

```
typeof(gatos[,1])
```

Output

```
[1] "character"
```

R

```
str(gatos[,1])
```

Output

```
 chr [1:3] "calico" "black" "tabby"
```

Cada fila es una *observación* de diferentes variables del mismo **data.frame**, y por lo tanto puede estar compuesto de elementos de diferentes tipos.

R

```
gatos[1,]
```

Output

```
    coat weight likes_string
1 calico    2.1            1
```

R

```
typeof(gatos[1,])
```

Output

```
[1] "list"
```

R

```
str(gatos[1,])
```

Output

```
'data.frame':   1 obs. of  3 variables:
 $ coat        : chr "calico"
 $ weight      : num 2.1
 $ likes_string: int 1
```

Desafío 3

Hay varias maneras sutilmente diferentes de indicar variables, observaciones y elementos de **data.frames**:

- `gatos[1]`
- `gatos[[1]]`
- `gatos$color`
- `gatos["color"]`
- `gatos[1, 1]`
- `gatos[, 1]`
- `gatos[1, ]`

Investiga cada uno de los ejemplos anteriores y explica el resultado de cada uno.

Sugerencia: Usa la función `typeof()` para examinar el resultado en cada caso.

Solución al desafío 3

Matrices

Por último, pero no menos importante, están las matrices. Podemos declarar una matriz llena de ceros de la siguiente forma:

R

```
matrix_example <- matrix(0, ncol=6, nrow=3)
matrix_example
```

Output

```
     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]    0    0    0    0    0    0
[2,]    0    0    0    0    0    0
[3,]    0    0    0    0    0    0
```

Y de manera similar a otras estructuras de datos, podemos preguntar cosas sobre la matriz:

R

```
class(matrix_example)
```

Output

```
[1] "matrix" "array"
```

R

```
typeof(matrix_example)
```

Output

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

R

```
str(matrix_example)
```

Output

```
 num [1:3, 1:6] 0 0 0 0 0 0 0 0 0 0 ...
```

R

```
dim(matrix_example)
```

Output

```
[1] 3 6
```

R

```
nrow(matrix_example)
```

Output

```
[1] 3
```

R

```
ncol(matrix_example)
```

Output

```
[1] 6
```

Desafío 4

¿Cuál crees que es el resultado del comando `length(matrix_example)` ? Inténtalo. ¿Estabas en lo correcto? ¿Por qué / por qué no?

Solución al desafío 4

Desafío 5

Construye otra matriz, esta vez conteniendo los números 1:50, con 5 columnas y 10 renglones. ¿Cómo llenó la función **`matrix`** de manera predeterminada la matriz, por columna o por renglón? Investiga como cambiar este comportamento. (Sugerencia: lee la documentación de la función **`matrix`** .)

Solución al desafío 5

Desafío 6

Crea una lista de longitud dos que contenga un vector de caracteres para cada una de las secciones en esta parte del curso:

- tipos de datos
- estructura de datos

Inicializa cada vector de caracteres con los nombres de los tipos de datos y estructuras de datos que hemos visto hasta ahora.

Solución al desafío 6

Desafío 7

Considera la salida de R para la siguiente matriz:

Output

```
     [,1] [,2]
[1,]    4    1
[2,]    9    5
[3,]   10    7
```

¿Cuál fué el comando correcto para escribir esta matriz? Examina cada comando e intenta determinar el correcto antes de escribirlos. Piensa en qué matrices producirán los otros comandos.

1. `matrix(c(4, 1, 9, 5, 10, 7), nrow = 3)`
2. `matrix(c(4, 9, 10, 1, 5, 7), ncol = 2, byrow = TRUE)`
3. `matrix(c(4, 9, 10, 1, 5, 7), nrow = 2)`
4. `matrix(c(4, 1, 9, 5, 10, 7), ncol = 2, byrow = TRUE)`

Solución al desafío 7

Puntos Clave

- Usar `read.csv` para leer los datos tabulares en R.
- Los tipos de datos básicos en R son **double**, **integer**, **complex**, **logical**, y **character**.
- Usar **factors** para representar categorías en R.

‹ (../03-seeking-help/index.html)

› (../05-data-structpart2/