ESTRUCTURAS DE DATOS EN

PERO ANTES...

guía de estilo de Google

otra: Hadley Wickham

validador: Google Rlint

lecturas sobre el validador...

OPERADOR DE ASIGNACIÓN

```
x <- 1
```

```
mes <- "Enero"
```

```
x <-  ## expresión incompleta: error
```

EVALUACIÓN

```
> x <- 5 ## no hay salida por pantalla
> x      ## impresión automática por pantalla
[1] 5
> print(x) ## impresión explícita por pantalla
[1] 5
```

¡ojo! no es lo mismo el objeto en R que su impresión por pantalla

OBJETOS: TIPOS "ATÓMICOS"

- carácter: `"Curso"`
- numérico (reales): `5.03`
- entero: `2`
- complejo: `2+4i`
- lógico: `TRUE/FALSE`

VECTORES

- tipo básico de objeto en R
- `vector()` crea un vector vacío
- `c()` combina sus argumentos para formar un vector

un vector sólo puede contener objetos del mismo tipo o clase (excepción: lista)

```
c("pepe",2,3)
c(1,2,3)
9:29
c(TRUE, FALSE)
vector("numeric", length = 10)
```

EJERCICIO

```
> library(swirl)
> swirl()
```

vectores

EJERCICIOS

crear los siguientes vectores:

1. (1, 2, 3, ..., 19, 20)
2. (20, 19, ..., 2, 1)
3. (1, 2, 3... 19, 20, 19, 28, ..., 2, 1)
4. (4, 6, 3) y asignárselo la variable `tmp`
5. (4, 6, 3, 4, 6, 3..., 4, 6, 3) con 10 ocurrencias de 4
6. (4, 6, 3, 4, 6, 3..., 4, 6, 3, 4) con 11 ocurrencias de 4 y 10 de 6 y 3
7. (4, 4...4, 6, 6..., 6, 3, 3..., 3) con 10 ocurrencias de 4, 20 de 6 y 30 de 3

usar la ayuda de la función rep() para 5, 6 y 7

EJERCICIOS

usar la función `paste()` para crear los siguientes vectores de caractéres de longitud 30:

1. ("etiqueta 1", "etiqueta 2", ..., "etiqueta 30").
2. ("fn1", "fn2", ..., "fn30").

ojo: en el primer caso hay una separación de un espacio en blanco; en el segundo, no

NÚMEROS

- objetos numéricos: números reales de precisión `double`
- entero explícito: `1L`
- `Inf`: infinito
- `NaN`: not a number (valores imposibles)
- `NA`: not available (valores indefinidos)

`is.na()`, `is.nan()` comprueban los objetos

un valor `NaN` también es `NA`, pero no al contrario

ATRIBUTOS

metadatos que describen el objeto

nombres de columnas, nombres de dimensiones, tamaño de matrices, clases, etc.

acceso usando la función `attributes()`

COERCIÓN

implícita:

```
> y <- c(1.7, "a") ## carácter
> y <- c(TRUE, 2)  ## numérico
> y <- c("a", TRUE)  ## carácter
```

explícita:

```
> x <- 0:6
> class(x)
[1] "integer"
> as.numeric(x)
[1] 0 1 2 3 4 5 6
> as.logical(x)
[1] FALSE TRUE TRUE TRUE TRUE TRUE TRUE
> as.character(x)
[1] "0" "1" "2" "3" "4" "5" "6"
```

MATRICES

son **vectores con un atributo de dimensión**, que a su vez es un vector de enteros de longitud 2 (número de filas y número de columnas)

```
> m <- matrix(nrow = 2, ncol = 3)
> m
     [,1] [,2] [,3]
[1,]   NA   NA   NA
[2,]   NA   NA   NA
> dim(m)
[1] 2 3
> attributes(m)
$dim
[1] 2 3
```

las matrices se construyen por columnas:

```
> m <- matrix(1:6, nrow = 2, ncol = 3)
> m
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
```

sólo pueden tener objetos de la misma clase

MATRICES

también se pueden construir a partir de vectores, añadiendo un atributo de dimensión:

```
> m <- 1:10
> m
[1] 1 2 3 4 5 6 7 8 9 10
> dim(m) <- c(2, 5)
> m
   [,1] [,2] [,3] [,4] [,5]
[1,]  1    3    5    7    9
[2,]  2    4    6    8   10
```

así como concatenando columnas o filas con `cbind()` o `rbind()`:

```
> x <- 1:3
> y <- 10:12
> cbind(x, y)
x y
[1,] 1 10
[2,] 2 11
[3,] 3 12
> rbind(x, y)
[,1] [,2] [,3]
x  1    2    3
y 10   11   12
```

EJERCICIO

dada la matriz

```
     [,1] [,2] [,3]
[1,]    1    1    3
[2,]    5    2    6
[2,]   -2   -1   -3
```

1. verificar que A * A * A es 0
2. reemplazar la tercera columna de A por la suma de la segunda y tercera columnas

LISTAS

tipo especial de vector que puede contener elementos de distintas clases

creación arbitraria con `list()`:

```
> x <- list(1, "a", TRUE, 1 + 4i)
> x
[[1]]
[1] 1
[[2]]
[1] "a"
[[3]]
[1] TRUE
[[4]]
[1] 1+4i
```

también se puede crear una lista vacía de tamaño predeterminado con `vector()`:

```
> x <- vector("list", length = 5)
```

FACTORES

utilizados para representar datos categóricos ordenados o no ordenados

un factor equivale a un vector de enteros donde cada entero tiene una **etiqueta**

son autodescriptivos

```
> x <- factor(c("Masculino", "Masculino", "Femenino", "Masculino", "Femenino"))
> x
[1] Masculino Masculino Femenino Masculino Femenino
Levels: Femenino Masculino
```

importante en problemas de modelado: **orden de los factores**

```
> x <- factor(c("si", "si", "no", "si", "no"), levels = c("si", "no"))
```

DATA FRAMES

importantísimo tipo de datos en R para almacenar datos en forma de **tabla**

se representan como un tipo especial de lista donde cada elemento de la lista tiene la misma longitud

dicho de otra forma:

cada elemento de la lista es una columna y la longitud de cada elemento es el número de filas

pueden contener objetos de distintas clases en cada columna

DATA FRAMES

atributos: nombres de columna `names()` y de fila `row.names()`

lo normal es que se creen a partir de `read.table()` o `read.csv()`

se pueden convertir a una matriz: `data.matrix()`

```
x <- data.frame(id = 1:4, edad = c(28, 15, 76, 23))
```

NOMBRES

los objetos en R pueden tener nombres, lo que permite escribir código legible y objetos auto-descriptivos vectores:

```
> x <- 1:3
> names(x) <- c("Cantabria", "España", "Total")
> x
```

listas:

```
> x <- list("Media" = 1, "Varianza" = 0, "Muestra" = 100)
> x
```

matrices:

```
> m <- matrix(1:4, nrow = 2, ncol = 2)
> dimnames(m) <- list(c("a", "b"), c("c", "d"))
> m
```

otra forma:

```
> colnames(m) <- c("h", "f")
> rownames(m) <- c("x", "z")
> m
```

NOMBRES

ojo: dataframes y matrices

Objeto	Definir nombres de columna	Definir nombres de fila
data frame	names()	row.names()
matriz	colnames()	rownames()

SUBSETTING

- operador `[`: devuelve objeto de la misma clase que el original. Se pueden seleccionar múltiples elementos de un objeto.
- operador `[[`: usado para extraer elementos de una lista o un data frame. Sólo se puede extraer un único elemento que no tiene por qué ser de tipo lista o data frame.
- operador $: usado para extraer elementos de una lista o un data frame por nombre literal. Su semántica es similar a `[[`.

SUBSETTING DE UN VECTOR

```
> x <- c("a", "b", "c", "c", "d", "a")
```

simple:

```
> x[2] ## Extraer el segundo elemento
[1] "b"
```

múltiple:

```
> x[1:4]
[1] "a" "b" "c" "c"
> x[c(1, 3, 4)] # con vector arbitrario
[1] "a" "c" "c"
```

que cumplen una condición lógica:

```
> x[x > "a"]
[1] "b" "c" "c" "d"
```

EJERCICIO

```
> library(swirl)
> swirl()
```

subsetting de vectores

SUBSETTING DE UNA MATRIZ

```
> x <- matrix(1:6, 2, 3)
```

simple:

```
> x[1, 2] ## fila 1, columna 2
[1] "3"
```

filas o columnas completas:

```
> x[1,] ## primera fila
[1] 1 3 5
> x[, 2] # segunda columna
[1] "3" "4"
```

SUBSETTING DE UNA MATRIZ

DIMENSION DROPPING

por defecto, cuando se recupera un único elemento de una matriz, este se devuelve en forma de **vector de un elemento** (y no de matriz de 1x1)

lo mismo ocurre al extraer una única fila o columna

para deshabilitar este comportamiento, usar `drop = FALSE`

```
> x <- matrix(1:6, 2, 3)
> x[1, 2]
[1] 3
> x[1, 2, drop = FALSE]
     [,1]
[1,]    3
```

SUBSETTING DE UNA LISTA

```
> x <- list(edad = 1:4, peso = 3.3)
> x
$edad
[1] 1 2 3 4
$peso
[1] 3.3
```

extracción de un único elemento:

```
> x[[1]]
[1] 1 2 3 4
> x[["peso"]] # con nombre de elemento
[1] 3.3
> x$peso # con operador $
[1] 3.3
```

el operador [[puede trabajar con índices calculados

el operador $ sólo con literales

```
> nombre <- "edad"
> x[[nombre]]
[1] 1 2 3 4
> x$nombre
NULL
```

SUBSETTING DE ELEMENTOS ANIDADOS EN UNA LISTA

el operador [[acepta una secuencia de enteros para extraer un elemento anidado en una lista

```
> x <- list(a = list(10, 12, 14), b = c(3.14, 2.81))
> x[[c(1, 3)]]
[1] 14
> ## de otra forma
> x[[1]][[3]]
[1] 14
```

¡ojo! no es lo mismo que en una matriz

SUBSETTING DE MÚLTIPLES ELEMENTOS EN UNA LISTA

```
> x <- list(cantidades = list(5, 6, 7), peso = 3.2, descripcion = "bultos")
> x[c(2,3)]
$peso
[1] 3.2
$descripcion
[1] "bultos"
```

¡ojo! no confundir `x[c(1,3)]` con `x[[c(1,3)]]`

COINCIDENCIAS PARCIALES

útil si el objeto de trabajo tiene nombres de elementos muy largos

utilizar sólo en modo interactivo, no en scripts largos

```
> x <- list(elementoMuyLargo = 1:5)
> x$e
[1] 1 2 3 4 5
> x[["e", exact = FALSE]]
[1] 1 2 3 4 5
```

ELIMINAR VALORES NO DISPONIBLES (NA)

```
> x <- c(1, 2, NA, 4, NA, 5)
> malos <- is.na(x)
> malos
[1] FALSE FALSE  TRUE FALSE  TRUE FALSE
> x[!malos]
[1] 1 2 4 5
```

caso de múltiples objetos:

```
> x <- c(1, 2, NA, 4, NA, 5)
> y <- c("a", "b", NA, "d", NA, "f")
> buenos <- complete.cases(x, y)
> buenos
[1] TRUE TRUE FALSE TRUE FALSE TRUE
> x[buenos]
[1] 1 2 4 5
> y[buenos]
[1] "a" "b" "d" "f"
```

`complete.cases` se puede usar con data frames

OPERACIONES VECTORIZADAS

muchas operaciones en R están **vectorizadas**: ocurren en paralelo

```
> x <- 1:4
> y <- 6:9
```

sin vectorización:

```
> z <- numeric(length(x))
> for(i in seq_along(x)) {
> z[i] <- x[i] + y[i]
> }
> z
> [1] 7 9 11 13
```

con vectorización:

```
> z <- x + y
> z
[1] 7 9 11 13
```

OPERACIONES VECTORIZADAS

vectorización de operación lógica:

```
> x
[1] 1 2 3 4
> x > 2
[1] FALSE FALSE TRUE TRUE
```

operaciones matriciales vectorizadas

multiplicación elemento a elemento:

```
> x <- matrix(1:4, 2, 2)
> y <- matrix(rep(10, 4), 2, 2)
> x * y
     [,1] [,2]
[1,]   10   30
[2,]   20   40
```

multiplicación matricial verdadera:

```
> x %*% y
     [,1] [,2]
[1,]   40   40
[2,]   60   60
```

LECTURA DE DATOS

`read.table()`: es conveniente leer la ayuda de la función con `?read.table`

para datasets de tamaño moderado:

```
> data <- read.table("dataset.txt")
```

Automáticamente:

- se ignoran las líneas con #
- se calcula el número de filas y se reserva espacio para ellas
- se averigua el tipo de variable de cada columna de la tabla

`read.csv()` es idéntica salvo por algunos parámetros por defecto (ej: separador)

LECTURA DE DATOS: DATASETS GRANDES

pasos previos para hacer nuestra vida más fácil:

- leer la página de ayuda para read.table()
- hacer una estimación de la memoria necesaria:

```
1.500.000 filas * 120 columnas * 8 bytes ~ 1.34 GB
# números doble precisión coma flotante
```

- usar `comment.char = ""` si no hay líneas comentadas
- usar el argumento `colClasses`: puede hacer que se ejecute **el doble de rápido**
- usar `nrows`: puede ayudar con el uso de memoria. `wc` en Unix calcula el número de filas

LECTURA DE DATOS: DATASETS GRANDES

ejemplo: obtener tipos o clases para cada columna:

```
> initial <- read.table("datatable.txt", nrows = 100)
> classes <- sapply(initial, class)
> tabAll <- read.table("datatable.txt", colClasses = classes)
```

LECTURA DE DATOS

paquete `readr`

- recientemente desarrollado por Hadley Wickham
- lectura de archivos planos grandes rápidamente
- funciones análogas: `read_table()` y `read_csv()` mucho más rápidas
- devuelve avisos si hay problemas no críticos durante la lectura
- lee archivos comprimidos

```
> library(readr)
> teams <- read_csv("data/team_standings.csv")
> teams
   Standing        Team
1         1       Spain
2         2 Netherlands
```

LECTURA DE DATOS

paquete `readr`

- la lectura del CSV es línea a línea
- los tipos de datos de las columnas se extraen de las primeras filas de datos
- la imputación de tipos de datos puede fallar
- se puede especificar el tipo de cada columna con `col_types`

```
> teams <- read_csv("data/team_standings.csv", col_types= "cc")
# cc abrevia carácter para ambas columnas
```

en general, es recomendable especificar los tipos explícitamente

LECTURA DE XLSX

```
> install.packages("xlsx")
> library(xlsx)
> datos <- read.xlsx("archivo.xlsx", sheetName="Hoja1")
```

LECTURA DE PC-AXIS

```
> install.packages("pxR")
> library(pxR)
> datos <- read.px("/archivo.px" )
```

ESCRITURA A CSV

```
## caso más sencillo
> write.csv(mis_datos, file = "mis_datos.csv")
## no incluir nombres de filas
> write.csv(mis_datos, file = "mis_datos.csv", row.names=FALSE)
## no incluir valores no disponibles
> write.csv(mis_datos, file = "mis_datos.csv", row.names=FALSE, na="")
## no exportar nombres de columna
> write.table(mis_datos, file = "mis_datos.csv", row.names=FALSE, na="",
              col.names=FALSE, sep=",")
```

ESCRITURA A XLSX

```
library(xlsx)
libro <- createWorkbook()
datos1 <- createSheet(wb=libro, sheetName="Datos 1")
datos2 <- createSheet(wb=libro, sheetName="Datos 2")
addDataFrame(x=df1, sheet=datos1)
addDataFrame(x=df2, sheet=datos22)
saveWorkbook(wb, "dataset.xlsx")
```

MÁS SOBRE DATAFRAMES

una observación por fila y una variable, medida o característica de dicha observación por columna

funciones de exploración:

- `dim()`: mostrar dimensiones
- `nrow()` y `ncol()`: mostrar número de filas y columnas
- `head()`: previsualizar las primeras filas del dataset
- `tail()`: previsualizar las últimas filas del dataset
- `summary()`: mostrar información sobre cada variable
- `str()`: mostrar estructura sobre el data frame. **Muy potente**

EJERCICIO

```
> library(swirl)
> swirl()
```

exploración de datos

OTRAS MEDIDAS DESCRIPTIVAS ELEMENTALES

- `mean()`: media
- `sd()`: desviación estándar

EJERCICIO

1. leer el archivo
   `http://verso.mat.uam.es/~joser.berrendero/datos/notas.t`
   con separador: espacio, separador decimal: coma, cabeceras: sí
2. ver los nombres de la variable del fichero
3. inspeccionar las primeras filas
4. obtener la media de ambos cursos
5. obtener la desviación estándar de ambos cursos
6. obtener un resumen de todas las variables del fichero

MÁS SOBRE DATAFRAMES

paquete `deplyr`, de Hadley Wickham (otra vez)

proporciona una gramática sencilla para trabajar con data frames

además, sus funciones son **muy rápidas** (codificadas en C++)

PROPIEDADES DE LAS FUNCIONES

- el primer argumento es un data frame
- el resto de argumentos indican qué hacer con él
- es posible hacer referencia a las columnas sin usar el operador $
- el resultado es un nuevo data frame
- los data frames deben estar ordenados: una observación por fila y una característica por columna

INSTALACIÓN Y USO

```
> install.packages("dplyr")
> library(dplyr)
```

PREPARACIÓN

```
> dataset <- read.px("http://www.icane.es/data/api/active-population-aged-16-more-gen
> datos <- dataset$DATA$value
> str(datos)
```

`select()`

```
> names(datos)
> subset <- select(datos, Sexo:value)
> head(subset)
```

normalmente, el operador `:` no se puede usar con nombres o cadenas, pero dentro de `select()` se puede utilizar para especificar un rango de variables

```
> select(datos, -(Sexo:value))
```

el signo negativo `-` permite exluir variables de la selección

así se haría sin `dplyr`:

```
> i <- match("Sexo", names(datos))
> j <- match("value", names(datos))
> head(datos[, -(i:j)])
```

`select()`

se pueden seleccionar nombres de variable basados en patrones:

```
> head(select(datos, ends_with("es")))
> head(select(datos, starts_with("S")))
```

`filter()`

utilizado para extraer subconjuntos de filas. Más rápida que `subset()`

```
> tail(filter(datos, Variables=="Activos" & Sexo=="Ambos sexos" & Grupo.de.edad=="Tot
> head(filter(datos, Variables=="Activos" & Sexo=="Ambos sexos" & Grupo.de.edad=="Tot
> filtrado <- filter(datos, Variables=="Activos" & Sexo=="Ambos sexos" & Grupo.de.eda
```

importante: la variable `filtrado` se usará a continuación

`arrange()`

utilizado para reordenar las filas de un data frame según una de las variables o columnas, preservando el orden del resto (**normalmente un dolor hacerlo en R**)

```
> arrange(filtrado, desc(value))
```

`rename()`

sorprendentemente, renombrar una columna de un data frame es complicado en R

```
> names(datos)[names(datos)=="value"] <- "Valor"
```

ahora ya no tanto

```
 > datos <- rename(datos, Valor = value)
```

`mutate()`

permite realizar transformaciones de variables en un data frame

```
mutate(filtrado, value.new = value - mean(value))
```

`transmute()` hace lo mismo, sólo que descarta las variables no transformadas

%>%

operador tubería: concatena varias funciones de dplyr en una sóla. sustituye a las expresiones anidadas del tipo:

```
tercera(segunda(primera(x)))
```

por:

```
primera(x) %>% segunda %>% tercera
```

INCISO

paquete `tidyr`: permite ordenar los datos

```
> install.packages("tidyr")
> library("tidyr");
```

`gather()`: datos "anchos" a "altos". Múltiples columnas a pares clave-valor.

`separate()`: separa dos variables unidas en una columna.

`spread()`: datos "altos" a "anchos". Dos columnas (par clave-valor) a múltiples columnas.

```
> filtrado <- filtrado %>% separate(Trimestres, c("anyo", "trimestre"), sep = " \\- "
```

`group_by()`

usado para generar estadísticas de resumen (agregados) dentro del estrato definido pro una variable

se suele utilizar con la función `summarize()` para agregar los resultados agrupados

```
> anyos <- group_by(filtrado, anyo)
> summarize(anyos, activos=mean(Valor, na.rm = TRUE))
```

EJERCICIO

data frame `iris` de ejemplo en R, con medidas del pétalo y sépalo de tres especies de lirios.

```
data(iris)
```

1. inspeccionar data frame
2. inspeccionar variable factor `Species`
3. averiguar qué índices corresponden a las 5 primeras observaciones de cada especie
4. hacer subsetting del data frame seleccionando las 5 primeras observaciones de cada especie en filas y todas las columnas, y almacenarlo en una variable `lirios`
5. mostrar el nuevo data frame por consola

EJERCICIO

selección de filas: `filter()`

1. lirios de la especie *setosa*
2. lirios de la especie *setosa* o *virginica*
3. lirios de la especie *setosa* con longitud de sépalo (`Sepal.Length`) inferior a 5 mm.

EJERCICIO

selección de columnas: `select()`

1. extraer longitud y ancho de sépalo para todas las observaciones
2. extraer todas las variables entre la longitud del pétalo y la del sépalo
3. extraer todas las variables menos la especie

EJERCICIO

ordenar: `arrange()`

1. ordenar de acuerdo con la longitud del sépalo, ascendente y descendente
2. ordenar según la especie por orden alfabético y luego de menor a mayor longitud del sépalo
3. extraer todas las variables menos la especie

EJERCICIO

sintaxis en cadena: %>%

seleccionar las variables que contienen las medidas del pétalo, los lirios para los que la longitud del pétalo es mayor que 4mm y ordenarlos de menor a mayor según la longitud del pétalo

EJERCICIO

añadir nuevas variables: `mutate()`

crear y añadir una variable `forma` a `lirios` que corresponda al cociente entre la anchura y la longitud del pétalo

usar el operador %>%

EJERCICIO

resumir subconjuntos de variables: `group_by()` + `summarise()`

calcular la media de la longitud del pétalo para los lirios de cada una de las especies

usar el operador %>%

Acceso al repositorio con la presentación

Acceso a la presentación

Fco. Javier Parra Rodríguez,Jefe de Servicio de Estadísticas Económicas y Socio-demográficas