Taller sobre el lenguaje R

Clase 1: Bases del lenguaje y tipos de datos

Lic. Lucio José Pantazis

March 17, 2021

Sobre el lenguaje R

Historia (S)

- Para hablar de R, primero hay que hablar de S.

Historia (S)

- Para hablar de R, primero hay que hablar de S.
- S es un lenguaje de programación desarrollado en 1976, utilizado por AT&T como software interno para análisis estadístico.

Historia (S)

- Para hablar de R, primero hay que hablar de S.
- S es un lenguaje de programación desarrollado en 1976, utilizado por AT&T como software interno para análisis estadístico.
- El objetivo inicial era obtener un lenguaje que permitiera hacer simple el análisis de datos, pero con posibilidades que sus usuarios puedan desarrollar programas extensos.

Historia (S)

- Para hablar de R, primero hay que hablar de S.
- S es un lenguaje de programación desarrollado en 1976, utilizado por AT&T como software interno para análisis estadístico.
- El objetivo inicial era obtener un lenguaje que permitiera hacer simple el análisis de datos, pero con posibilidades que sus usuarios puedan desarrollar programas extensos.
- Luego la empresa vendió la licencia para que pueda ser comercializado y pasó a llamarse S-PLUS.

Historia (R)

- Si bien R fue desarrollado en 1991, la primer publicación científica aparece en 1996 y se incorpora a la licencia pública de GNU. Se libera al público en el año 2000 por primera vez y pasa a ser software libre.

Historia (R)

- Si bien R fue desarrollado en 1991, la primer publicación científica aparece en 1996 y se incorpora a la licencia pública de GNU. Se libera al público en el año 2000 por primera vez y pasa a ser software libre.

- R surge como una alternativa más económica a S-PLUS, tratando de captar a los usuarios de S-PLUS y por lo tanto, manteniendo algunas expresiones horribles como la asignación:

```
x<-1
paste("x =",x)
```

```
## [1] "x = 1"
```

Esta notación aparece en muchos libros ya que es la asignación original del lenguaje S.

Historia (R)

- Si bien R fue desarrollado en 1991, la primer publicación científica aparece en 1996 y se incorpora a la licencia pública de GNU. Se libera al público en el año 2000 por primera vez y pasa a ser software libre.

- R surge como una alternativa más económica a S-PLUS, tratando de captar a los usuarios de S-PLUS y por lo tanto, manteniendo algunas expresiones horribles como la asignación:

```
x<-1
paste("x =",x)
```

```
## [1] "x = 1"
```

Esta notación aparece en muchos libros ya que es la asignación original del lenguaje S.

- De todos modos, R ya acepta la asignación mediante un símbolo de igualdad.

```
x=1
x
```

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

Historia (R)

- Otra expresión molesta es la forma que tiene R de concatenar elementos.

```
y=c(x,2)
y
```

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

```
z=c(y,sqrt(5))
z
```

```
## [1] 1.000000 2.000000 2.236068
```

Pros & Cons

- Ventajas:

Pros & Cons

- Ventajas:
  - Open Source

Pros & Cons

- Ventajas:
  - Open Source
  - R tiene una gran variedad de procedimientos estadísticos muy específicos.

Pros & Cons

- Ventajas:
  - Open Source
  - R tiene una gran variedad de procedimientos estadísticos muy específicos.
  - R maneja muy bien bases de datos e interpreta correctamente variables cualitativas a la hora de realizar cálculos.

Pros & Cons

- Ventajas:
  - Open Source
  - R tiene una gran variedad de procedimientos estadísticos muy específicos.
  - R maneja muy bien bases de datos e interpreta correctamente variables cualitativas a la hora de realizar cálculos.
  - Tiene capacidad de manipular una gran diversidad de elementos gráficos.

Pros & Cons

- Ventajas:
  - Open Source
  - R tiene una gran variedad de procedimientos estadísticos muy específicos.
  - R maneja muy bien bases de datos e interpreta correctamente variables cualitativas a la hora de realizar cálculos.
  - Tiene capacidad de manipular una gran diversidad de elementos gráficos.
- Desventajas

Pros & Cons

- Ventajas:
  - Open Source
  - R tiene una gran variedad de procedimientos estadísticos muy específicos.
  - R maneja muy bien bases de datos e interpreta correctamente variables cualitativas a la hora de realizar cálculos.
  - Tiene capacidad de manipular una gran diversidad de elementos gráficos.
- Desventajas
  - R utiliza la memoria física de una computadora. Puede consumir mucha memoria.

Pros & Cons

- Ventajas:
  - Open Source
  - R tiene una gran variedad de procedimientos estadísticos muy específicos.
  - R maneja muy bien bases de datos e interpreta correctamente variables cualitativas a la hora de realizar cálculos.
  - Tiene capacidad de manipular una gran diversidad de elementos gráficos.
- Desventajas
  - R utiliza la memoria física de una computadora. Puede consumir mucha memoria.
  - No tiene tanto nivel de precisión.

Paquetes

R tiene una cosa muy buena y es que tiene muchos paquetes.

Paquetes

R tiene una cosa muy buena y es que tiene muchos paquetes.

- Por ser Open Source, cualquier persona puede aportar paquetes. Por lo tanto, muchas herramientas ya están implementadas y se pueden instalar con el comando install.packages:

```
install.packages("ggplot2")
```

Paquetes

R tiene una cosa muy buena y es que tiene muchos paquetes.

- Por ser Open Source, cualquier persona puede aportar paquetes. Por lo tanto, muchas herramientas ya están implementadas y se pueden instalar con el comando install.packages:

```
install.packages("ggplot2")
```

- Una vez instalado, se carga con el comando library o require (permite no volver a cargar el paquete una vez cargado):

```
library(ggplot2)
require(ggplot2)
```

Paquetes

R tiene una cosa mala y es que tiene muchos paquetes.

- Muchas veces el nombre de una misma función puede estar en varios paquetes y hay que especificar el paquete que se utiliza, utilizando dos veces los dos puntos:

```
args(MASS::select)
```

```
## function (obj)
## NULL
```

```
args(dplyr::select)
```

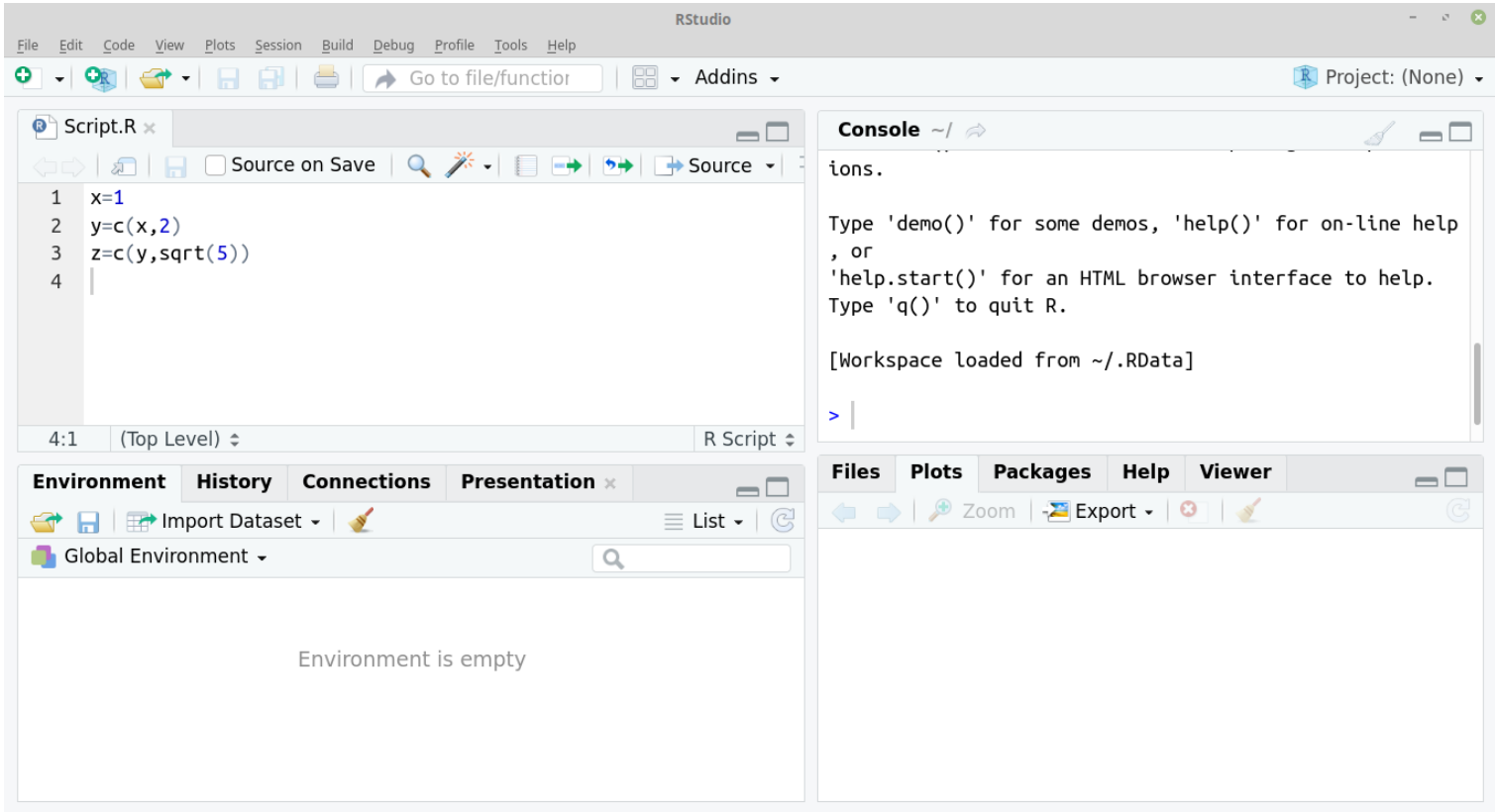
```
## function (.data, ...)
## NULL
```

CRAN

- El paquete más importante es el paquete base, y es lo primero que debe instalarse.
- Se consigue en la página <http://cran.r-project.org>.
- CRAN viene de "Comprehensive R Archive Network" y contiene documentación sobre gran parte de los paquetes implementados.

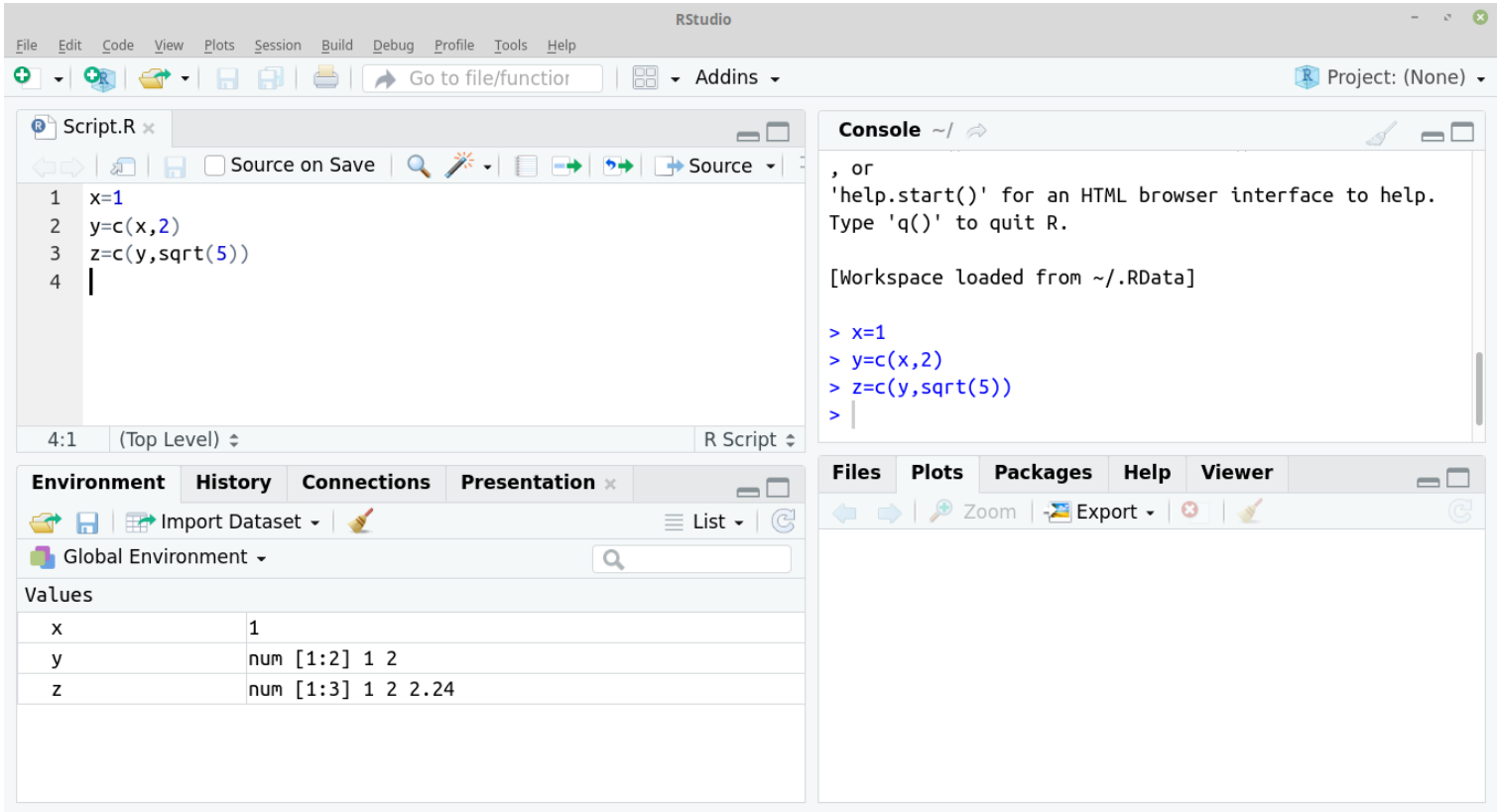
RStudio

Para trabajar en R, la interfaz RStudio es muy útil ya que permite ver varias cosas en simultáneo.



RStudio

Corriendo el script, van apareciendo las variables nuevas en el environment.



Tipos de datos

Vectores básicos o atómicos

Una cosa buena de R es que tiene muchos tipos de datos:

- **character:** strings de caracteres.

```
str1="ABC"
str1
```

```
## [1] "ABC"
```

```
class(str1)
```

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

Vectores básicos o atómicos

Una cosa buena de R es que tiene muchos tipos de datos:

- **character:** strings de caracteres.

```
str2="Lucio dejá de dar vueltas"
str2
```

```
## [1] "Lucio dejá de dar vueltas"
```

```
class(str2)
```

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

Vectores básicos o atómicos

Una cosa buena de R es que tiene muchos tipos de datos:

- **character:** strings de caracteres.

```
str3="Oblígame"
str3
```

```
## [1] "Oblígame"
```

```
class(str3)
```

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

Vectores básicos o atómicos

Una cosa buena de R es que tiene muchos tipos de datos:

- **character:** strings de caracteres.
- **numeric:** Números "reales"

```
num1=sqrt(5)
num1
```

```
## [1] 2.236068
```

```
class(num1)
```

```
## [1] "numeric"
```

Vectores básicos o atómicos

Una cosa buena de R es que tiene muchos tipos de datos:

- **character:** strings de caracteres.
- **numeric:** Números "reales"

```
num2=exp(1)
num2
```

```
## [1] 2.718282
```

```
class(num2)
```

```
## [1] "numeric"
```

Vectores básicos o atómicos

Una cosa buena de R es que tiene muchos tipos de datos:

- **character:** strings de caracteres.
- **numeric:** Números "reales"

```
num3=2
num3
```

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

```
class(num3)
```

```
## [1] "numeric"
```

Vectores básicos o atómicos

Una cosa buena de R es que tiene muchos tipos de datos:

- **character:** strings de caracteres.
- **numeric:** Números "reales".
- **integer:** Números enteros.

```
int1=2L
int1
```

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

```
class(int1)
```

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

Vectores básicos o atómicos

Una cosa buena de R es que tiene muchos tipos de datos:

- **character:** strings de caracteres.
- **numeric:** Números "reales"
- **integer:** Números enteros
- **complex:** Números complejos

```
comp1=1+1i
comp1
```

```
## [1] 1+1i
```

```
class(comp1)
```

```
## [1] "complex"
```

Vectores básicos o atómicos

Una cosa buena de R es que tiene muchos tipos de datos:

- **character:** strings de caracteres.
- **numeric:** Números "reales"
- **integer:** Números enteros
- **complex:** Números complejos

```
# Si no se agrega el coeficiente imaginario,
# considera i como una variable
comp2=1+i
```

```
## Error in eval(expr, envir, enclos): object 'i' not found
```

Vectores básicos o atómicos

Una cosa buena de R es que tiene muchos tipos de datos:

- **character:** strings de caracteres.
- **numeric:** Números "reales"
- **integer:** Números enteros
- **complex:** Números complejos
- **logical:** TRUE/FALSE

```
log1=2>1
log1
```

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

```
class(log1)
```

```
## [1] "logical"
```

Vectores básicos o atómicos

Una cosa buena de R es que tiene muchos tipos de datos:

- **character:** strings de caracteres.
- **numeric:** Números "reales"
- **integer:** Números enteros
- **complex:** Números complejos
- **logical:** TRUE/FALSE

```
log2=1==0
log2
```

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

```
class(log2)
```

```
## [1] "logical"
```

Vectores básicos o atómicos

Una cosa buena de R es que tiene muchos tipos de datos:

- **character:** strings de caracteres.
- **numeric:** Números "reales"
- **integer:** Números enteros
- **complex:** Números complejos
- **logical:** TRUE/FALSE

```
log3= 2!=0
log3
```

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

```
class(log3)
```

```
## [1] "logical"
```

Vectores básicos o atómicos

Una cosa buena de R es que tiene muchos tipos de datos:

- **character:** strings de caracteres.
- **numeric:** Números "reales"
- **integer:** Números enteros
- **complex:** Números complejos
- **logical:** TRUE/FALSE

Vectores básicos o atómicos

Una cosa buena de R es que tiene muchos tipos de datos:

- **character:** strings de caracteres.
- **numeric:** Números "reales"
- **integer:** Números enteros
- **complex:** Números complejos
- **logical:** TRUE/FALSE

Sin embargo,

Vectores básicos o atómicos

Una cosa buena de R es que tiene muchos tipos de datos:

- **character:** strings de caracteres.
- **numeric:** Números "reales"
- **integer:** Números enteros
- **complex:** Números complejos
- **logical:** TRUE/FALSE

Sin embargo, una cosa mala de R es que tiene muchos tipos de datos.

Vectores básicos o atómicos

Una cosa buena de R es que tiene muchos tipos de datos:

- **character:** strings de caracteres.
- **numeric:** Números "reales"
- **integer:** Números enteros
- **complex:** Números complejos
- **logical:** TRUE/FALSE

Sin embargo, una cosa mala de R es que tiene muchos tipos de datos. A veces podemos pensar que las variables están en un tipo de datos y no el esperado.

Coerción

Muchas veces, R convierte un tipo de dato a otro para poder realizar la instrucción. (Ojo: No siempre avisa el pillín)

Coerción

Muchas veces, R convierte un tipo de dato a otro para poder realizar la instrucción. (Ojo: No siempre avisa el pillín)

- Cualquier dato se puede convertir en un caracter añadiendo comillas. Por lo tanto, es la coerción más común.

```
x=1
coe1=paste0("x=",x)
coe1
```

```
## [1] "x=1"
```

```
class(coe1)
```

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

Coerción

Muchas veces, R convierte un tipo de dato a otro para poder realizar la instrucción. (Ojo: No siempre avisa el pillín)

- Cualquier dato se puede convertir en un caracter añadiendo comillas. Por lo tanto, es la coerción más común.

```
coe2="2"
class(coe2)
```

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

Coerción

Muchas veces, R convierte un tipo de dato a otro para poder realizar la instrucción. (Ojo: No siempre avisa el pillín)

- Cualquier dato se puede convertir en un caracter añadiendo comillas. Por lo tanto, es la coerción más común.
- Ante cualquier operación con una variable numérica, las variables enteras y lógicas pasan a ser numéricas.

```
int1=2L
num1=sqrt(5)
coe3=num1+int1
coe3
```

```
## [1] 4.236068
```

```
class(coe3)
```

```
## [1] "numeric"
```

Coerción

Muchas veces, R convierte un tipo de dato a otro para poder realizar la instrucción. (Ojo: No siempre avisa el pillín)

- Cualquier dato se puede convertir en un caracter añadiendo comillas. Por lo tanto, es la coerción más común.
- Ante cualquier operación con una variable numérica, las variables enteras y lógicas pasan a ser numéricas.

```
log1=2>1
num1=sqrt(5)
coe4=num1+log1
coe4
```

```
## [1] 3.236068
```

```
class(coe4)
```

```
## [1] "numeric"
```

Coerción

Muchas veces, R convierte un tipo de dato a otro para poder realizar la instrucción. (Ojo: No siempre avisa el pillín)

- Cualquier dato se puede convertir en un caracter añadiendo comillas. Por lo tanto, es la coerción más común.
- Ante cualquier operación con una variable numérica, las variables enteras y lógicas pasan a ser numéricas.
- A veces, no sabe como coercionar y tira un error. Por ejemplo, si se quieren sumar un caracter con un número.

```
str1="ABC"
num1=sqrt(5)
coe5=num1+str1
```

```
## Error in num1 + str1: non-numeric argument to binary operator
```

Coerción Explícita

Hay comandos que convierten un tipo de dato en otro de forma explícita. Tienen la estructura "as._____", donde se reemplaza el espacio en blanco por un tipo de dato. Ejemplos:

```
log1=2>1
as.integer(log1)
```

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

```
as.numeric(log1)
```

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

```
as.character(log1)
```

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

```
as.complex(log1)
```

```
## [1] 1+0i
```

Coerción Explícita

Hay comandos que convierten un tipo de dato en otro de forma explícita. Tienen la estructura "as._____", donde se reemplaza el espacio en blanco por un tipo de dato. Ejemplos:

```
str1="ABC"
str3="2"
as.numeric(str1)
```

```
## Warning: NAs introduced by coercion
```

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

```
as.numeric(str3)
```

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

Datos Faltantes

Cuando R tiene que llenar una estructura con datos que no sabe cómo determinar, reemplaza por los siguientes valores:

- NA: "No answer"

```
str1="ABC"
as.numeric(str1)
```

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

Datos Faltantes

Cuando R tiene que llenar una estructura con datos que no sabe cómo determinar, reemplaza por los siguientes valores:

- NA: "No answer"
- NaN: "Not a number"

```
ind1=0/0
as.numeric(ind1)
```

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

Datos Faltantes

Cuando R tiene que llenar una estructura con datos que no sabe cómo determinar, reemplaza por los siguientes valores:

- NA: "No answer"
- NaN: "Not a number"
- Inf: "Infinity"

```
inf1=1/0
as.numeric(inf1)
```

```
## [1] Inf
```

Verificación

Hay comandos que verifican si un objeto pertenece a un cierto tipo de dato. Son variables lógicas y tienen la estructura "is._____", donde se reemplaza el espacio en blanco por el tipo de dato.

```
str1="ABC"
is.numeric(str1)
```

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

```
is.character(str1)
```

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

Verificación

Hay comandos que verifican si un objeto pertenece a un cierto tipo de dato. Son variables lógicas y tienen la estructura "is._____", donde se reemplaza el espacio en blanco por el tipo de dato.

```
str1="ABC"
coe6=as.numeric(str1)
is.numeric(coe6)
```

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

```
is.character(coe6)
```

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

```
is.na(coe6)
```

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

Verificación

Hay comandos que verifican si un objeto pertenece a un cierto tipo de dato. Son variables lógicas y tienen la estructura "is._____", donde se reemplaza el espacio en blanco por el tipo de dato.

```
str3="2"
is.numeric(str3)
```

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

```
is.character(str3)
```

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

Verificación

Hay comandos que verifican si un objeto pertenece a un cierto tipo de dato. Son variables lógicas y tienen la estructura "is._____", donde se reemplaza el espacio en blanco por el tipo de dato.

```
str3="2"
coe7=as.numeric(str3)
is.numeric(coe7)
```

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

```
is.character(coe7)
```

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

```
is.na(coe7)
```

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

Vectores

Ya vimos que una forma de armar vectores es concatenando:

```
vec1=c(1,4,8)
vec1
```

```
## [1] 1 4 8
```

Vectores

Cada vector tiene un único tipo de datos, por lo que si se unen distintos tipos de datos, se coercionan a una única clase:

```
vec2=c("ABC",1+1i,2)
vec2
```

```
## [1] "ABC"  "1+1i" "2"
```

```
class(vec2)
```

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

Vectores

Cada vector tiene un único tipo de datos, por lo que si se unen distintos tipos de datos, se coercionan a una única clase:

```
vec3=c(sqrt(5),4L,2<1)
vec3
```

```
## [1] 2.236068 4.000000 0.000000
```

```
class(vec3)
```

```
## [1] "numeric"
```

Vectores

La coerción también ocurre al operar:

```
vec1=c(1L,4L,8L)
vec3=c(sqrt(5),4L,2<1)
vec4=vec1*vec3
vec4
```

```
## [1]  2.236068 16.000000  0.000000
```

```
class(vec1)
```

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

```
class(vec3)
```

```
## [1] "numeric"
```

```
class(vec4)
```

```
## [1] "numeric"
```

Vectores

En el caso de tener distintas longitudes, multiplica el vector más corto coordenada a coordenada hasta que se acaba y vuelve a repetir.

```
vec1=c(1L,4L) # length=2
vec2=c(-1,2,3) # length=3
vec3=c(-2,2,5,8L)# length=4
vec4=vec1*vec2
```

```
## Warning in vec1 * vec2: longer object length is not a multiple of shorter
## length
```

```
vec5=vec1*vec3
vec4
```

```
## [1] -1  8  3
```

```
vec5
```

```
## [1] -2  8  5 32
```

Vectores

Para hacer el producto escalar, hay que recurrir a otra de las notaciones nefastas que quedaron del lenguaje S:

```
vec1=c(1L,4L,8L)
vec3=c(sqrt(5),4L,2<1)
vec4=vec1%*%vec3
1*sqrt(5)+4*4+0*8
```

```
## [1] 18.23607
```

```
vec4
```

```
##          [,1]
## [1,] 18.23607
```

Vectores

Para armar vectores con una única componente se usa el comando rep:

```
rep(2,6)
```

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

Sin embargo, se puede usar para repetir un vector una cantidad de veces

```
rep(c("A","B"),3)
```

```
## [1] "A" "B" "A" "B" "A" "B"
```

```
rep(c("A","B"),3,each=2)
```

```
##  [1] "A" "A" "B" "B" "A" "A" "B" "B" "A" "A" "B" "B"
```

Vectores

Para armar secuencias, se pueden armar con saltos unitarios como en varios lenguajes:

```
-5:10
```

```
##  [1] -5 -4 -3 -2 -1  0  1  2  3  4  5  6  7  8  9 10
```

Para armar secuencias con saltos específicos, se usa el comando seq:

```
seq(2,3,0.1)
```

```
##  [1] 2.0 2.1 2.2 2.3 2.4 2.5 2.6 2.7 2.8 2.9 3.0
```

```
seq(2,3,0.3)
```

```
## [1] 2.0 2.3 2.6 2.9
```

Matrices

Para definir matrices, (oh sorpresa!) se usa el comando matrix. Primero recibe los datos en un vector, y luego el número de filas y columnas. Ejemplos:

```
Mat1=matrix(1:6,2,3)
Mat2=matrix(1:6,3,2)
Mat1
```

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

```
Mat2
```

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

Matrices

Para definir matrices, (oh sorpresa!) se usa el comando matrix. Primero recibe los datos en un vector, y luego el número de filas y columnas. Ejemplos:

```
Mat3=matrix(1:6,2,3,byrow = TRUE)
Mat4=matrix(1:6,3,2,byrow = TRUE)
Mat3
```

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

```
Mat4
```

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

Matrices

En el caso en que la longitud del vector es mayor que el producto de filas y columnas, rellena la matriz hasta donde puede. En caso de que el vector sea de menor longitud, llena la matriz hasta acabarse y luego vuelve a empezar.

Ejemplos:

```
Mat5=matrix(1:8,2,3)
Mat6=matrix(1:2,3,2)
Mat5
```

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

```
Mat6
```

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

Matrices

Para transponer se usa el comando t:

```
Mat1=matrix(1:6,2,3)
Mat1
```

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

```
t(Mat1)
```

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

Matrices

Al igual que los vectores, las matrices tienen un único tipo de datos y se coercionan:

```
Mat7=matrix(c(1,"A",3,4,sqrt(5),6),2,3)
Mat7
```

```
##      [,1] [,2] [,3]
## [1,] "1"  "3"  "2.23606797749979"
## [2,] "A"  "4"  "6"
```

```
class(Mat7)
```

```
## [1] "matrix"
```

```
typeof(Mat7)
```

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

Matrices

Al igual que los vectores, las matrices primero multiplican coordenada a coordenada. En este caso, sí deben tener las mismas dimensiones

```
Mat1=matrix(1:6,2,3)
Mat2=matrix(1:6,3,2)
Mat1*Mat2
```

```
## Error in Mat1 * Mat2: non-conformable arrays
```

```
Mat1*t(Mat2)
```

```
##      [,1] [,2] [,3]
## [1,]    1    6   15
## [2,]    8   20   36
```

Matrices

Para el producto matricial, se usa la misma sintaxis que para el producto escalar de vectores. Las dimensiones deben coordinar como el producto usual de matrices.

```
Mat1=matrix(1:6,2,3)
Mat2=matrix(1:6,3,2)
Mat1%*%Mat2
```

```
##      [,1] [,2]
## [1,]   22   49
## [2,]   28   64
```

```
Mat1%*%t(Mat2)
```

```
## Error in Mat1 %*% t(Mat2): non-conformable arguments
```

Matrices

R tiene dos operaciones útiles para concatenar matrices por filas o columnas, llamadas rbind y cbind respectivamente. Obviamente, deben tener las dimensiones correctas:

```
Mat1=matrix(1:6,2,3)
Mat3=matrix(1:6,2,3,byrow = T)
rbind(Mat1,Mat3)
```

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

```
rbind(Mat1,t(Mat3))
```

```
## Error in rbind(Mat1, t(Mat3)): number of columns of matrices must match
```

Matrices

R tiene dos operaciones útiles para concatenar matrices por filas o columnas, llamadas rbind y cbind respectivamente. Obviamente, deben tener las dimensiones correctas:

```
Mat1=matrix(1:6,2,3)
Mat3=matrix(1:6,2,3,byrow = T)
cbind(Mat1,Mat3)
```

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

```
cbind(Mat1,t(Mat3))
```

```
## Error in cbind(Mat1, t(Mat3)): number of rows of matrices must match (see
```

Matrices

Además, ya tiene funciones para sumar y tomar promedios por fila o columna:

```
Mat1=matrix(1:6,2,3)
Mat1
```

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

```
rowSums(Mat1)
```

```
## [1]  9 12
```

```
rowMeans(Mat1)
```

```
## [1] 3 4
```

Matrices

Además, ya tiene funciones para sumar y tomar promedios por fila o columna:

```
Mat1=matrix(1:6,2,3)
Mat1
```

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

```
colSums(Mat1)
```

```
## [1]  3  7 11
```

```
colMeans(Mat1)
```

```
## [1] 1.5 3.5 5.5
```

Listas

Las listas permiten guardar distintos tipos de datos, sin coerción.

```
Lis1=list("a",1,TRUE)
Lis1
```

```
## [[1]]
## [1] "a"
##
## [[2]]
## [1] 1
##
## [[3]]
## [1] TRUE
```

```
typeof(Lis1)
```

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

Listas

Se agregan nuevos elementos a la lista con el comando append.

```
Lis1=list("a",1,TRUE)
Lis2=append(Lis1,1+1i)
Lis2
```

```
## [[1]]
## [1] "a"
##
## [[2]]
## [1] 1
##
## [[3]]
## [1] TRUE
##
## [[4]]
## [1] 1+1i
```

Nombrando elementos

R tiene el comando names para darle un nombre a cada elemento de un vector o lista. Los nombres deben estar en caracteres

```
vec1=c(1L,4L,8L)
names(vec1)=c("Bart","Lisa","Maggie")
vec1
```

```
##   Bart   Lisa Maggie
##      1      4      8
```

Nombrando elementos

R tiene el comando names para darle un nombre a cada elemento de un vector o lista. Los nombres deben estar en caracteres

```
Lis1=list("a",1,TRUE)
names(Lis1)=c("Moe","Larry","Curly")
Lis1
```

```
## $Moe
## [1] "a"
##
## $Larry
## [1] 1
##
## $Curly
## [1] TRUE
```

Nombrando elementos

En el caso de tener menos nombres que componentes de un vector, llena los nombres restantes con NA.

```
vec1=c(1L,4L,8L)
names(vec1)=c("Homero","Marge")
vec1
```

```
## Homero  Marge   <NA>
##      1      4      8
```

Nombrando elementos

En el caso de tener menos nombres que componentes de una lista, llena los nombres restantes con NA.

```
Lis1=list("a",1,TRUE)
names(Lis1)=c("Gardel","LePera")
Lis1
```

```
## $Gardel
## [1] "a"
##
## $LePera
## [1] 1
##
## $<NA>
## [1] TRUE
```

Nombrando elementos

Otra forma es agregar los nombres en la asignación del vector del siguiente modo:

```
vec1=c(Bart=1L,Lisa=4L,Maggie=8L)
vec1
```

```
##   Bart   Lisa Maggie
##      1      4      8
```

Notar que no hace falta agregar caracteres al ser una cuestión interna de la asignación.

Nombrando elementos

Del mismo modo también se pueden asignar los nombres de una lista:

```
Lis1=list(Moe="a",Larry=1,Curly=TRUE)
Lis1
```

```
## $Moe
## [1] "a"
##
## $Larry
## [1] 1
##
## $Curly
## [1] TRUE
```

Nombrando elementos

En el caso de tener matrices, se usan los comandos rownames y colnames para especificar cuál de las dimensiones se nombran.

```
Mat1=matrix(1:6,2,3)
rownames(Mat1)=c("Jagger","Richards")
colnames(Mat1)=c("Wood","Watts","Jones")
Mat1
```

```
##           Wood Watts Jones
## Jagger       1     3     5
## Richards     2     4     6
```

Nombrando elementos

En el caso no coincidir alguna dimensión tira error:

```
Mat1=matrix(1:6,2,3)
Mat1
```

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

```
rownames(Mat1)=c("Lennon","McCartney")
colnames(Mat1)=c("Harrison","Starr")
```

```
## Error in dimnames(x) <- dn: length of 'dimnames' [2] not equal to array e
```

Subconjuntos

R tiene mucha versatilidad a la hora de buscar coordenadas y subconjuntos de vectores, matrices y listas.

- Puede ser con vectores: Selecciona las componentes del vector.

```
vec1=c(Bart=1L,Lisa=4L,Maggie=8L)
vec1[c(1,3)]
```

```
##   Bart Maggie
##      1      8
```

Subconjuntos

R tiene mucha versatilidad a la hora de buscar coordenadas y subconjuntos de vectores, matrices y listas.

- Puede ser con vectores: Selecciona las componentes del vector.

```
Lis1=list(Moe="a",Larry=1,Curly=TRUE)
Lis1[3:2]
```

```
## $Curly
## [1] TRUE
##
## $Larry
## [1] 1
```

Subconjuntos

R tiene mucha versatilidad a la hora de buscar coordenadas y subconjuntos de vectores, matrices y listas.

- Puede ser con vectores: Selecciona las componentes del vector.

```
Mat1
```

```
##          Wood Watts Jones
## Jagger      1     3     5
## Richards    2     4     6
```

```
Mat1[2,1:2]
```

```
##  Wood Watts
##     2     4
```

Subconjuntos

R tiene mucha versatilidad a la hora de buscar coordenadas y subconjuntos de vectores, matrices y listas.

- Puede ser con vectores: Selecciona las componentes del vector.
- Puede ser con espacios blancos: en matrices, permite elegir filas o columnas enteras

```
Mat1
```

```
##          Wood Watts Jones
## Jagger      1     3     5
## Richards    2     4     6
```

```
Mat1[,rep(1,2)]
```

```
##          Wood Wood
## Jagger      1    1
## Richards    2    2
```

Subconjuntos

R tiene mucha versatilidad a la hora de buscar coordenadas y subconjuntos de vectores, matrices y listas.

- Puede ser con vectores: Selecciona las componentes del vector.
- Puede ser con espacios blancos: en matrices, permite elegir filas o columnas enteras.
- Puede ser con valores negativos: Elimina las componentes seleccionadas.

```
vec1=c(Bart=1L,Lisa=4L,Maggie=8L)
vec1[-1]
```

```
##   Lisa Maggie
##      4      8
```

Subconjuntos

R tiene mucha versatilidad a la hora de buscar coordenadas y subconjuntos de vectores, matrices y listas.

- Puede ser con vectores: Selecciona las componentes del vector.
- Puede ser con espacios blancos: en matrices, permite elegir filas o columnas enteras.
- Puede ser con valores negativos: Elimina las componentes seleccionadas.

```
Lis1=list(Moe="a",Larry=1,Curly=TRUE)
Lis1[-(1:2)]
```

```
## $Curly
## [1] TRUE
```

Subconjuntos

R tiene mucha versatilidad a la hora de buscar coordenadas y subconjuntos de vectores, matrices y listas.

- Puede ser con vectores: Selecciona las componentes del vector.
- Puede ser con espacios blancos: en matrices, permite elegir filas o columnas enteras.
- Puede ser con valores negativos: Elimina las componentes seleccionadas.

```
Mat1
```

```
##          Wood Watts Jones
## Jagger      1     3     5
## Richards    2     4     6
```

```
Mat1[,-2]
```

```
##          Wood Jones
## Jagger      1     5
## Richards    2     6
```

Subconjuntos

R tiene mucha versatilidad a la hora de buscar coordenadas y subconjuntos de vectores, matrices y listas.

- Puede ser con vectores: Selecciona las componentes del vector.
- Puede ser con espacios blancos: en matrices, permite elegir filas o columnas enteras.
- Puede ser con valores negativos: Elimina las componentes seleccionadas.
- Puede ser con los nombres de las variables:

```
vec1=c(Bart=1L,Lisa=4L,Maggie=8L)
vec1[c("Lisa","Maggie")]
```

```
##   Lisa Maggie
##      4      8
```

Subconjuntos

R tiene mucha versatilidad a la hora de buscar coordenadas y subconjuntos de vectores, matrices y listas.

- Puede ser con vectores: Selecciona las componentes del vector.
- Puede ser con espacios blancos: en matrices, permite elegir filas o columnas enteras.
- Puede ser con valores negativos: Elimina las componentes seleccionadas.
- Puede ser con los nombres de las variables:

```
Lis1=list(Moe="a",Larry=1,Curly=TRUE)
Lis1["Moe"]
```

```
## $Moe
## [1] "a"
```

Subconjuntos

R tiene mucha versatilidad a la hora de buscar coordenadas y subconjuntos de vectores, matrices y listas.

- Puede ser con vectores: Selecciona las componentes del vector.
- Puede ser con espacios blancos: en matrices, permite elegir filas o columnas enteras.
- Puede ser con valores negativos: Elimina las componentes seleccionadas.
- Puede ser con los nombres de las variables:

```
Mat1
```

```
##          Wood Watts Jones
## Jagger      1     3     5
## Richards    2     4     6
```

```
Mat1["Jagger",c("Watts","Wood"),drop=F]
```

```
##        Watts Wood
## Jagger     3    1
```

Subconjuntos

R tiene mucha versatilidad a la hora de buscar coordenadas y subconjuntos de vectores, matrices y listas.

- Puede ser con vectores: Selecciona las componentes del vector.
- Puede ser con espacios blancos: en matrices, permite elegir filas o columnas enteras.
- Puede ser con valores negativos: Elimina las componentes seleccionadas.
- Puede ser con los nombres de las variables.
- Puede ser con variables lógicas

```
Log1=1:2<2;Log2=1:3>1
Log1;Log2
```

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

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

Subconjuntos

R tiene mucha versatilidad a la hora de buscar coordenadas y subconjuntos de vectores, matrices y listas.

- Puede ser con vectores: Selecciona las componentes del vector.
- Puede ser con espacios blancos: en matrices, permite elegir filas o columnas enteras.
- Puede ser con valores negativos: Elimina las componentes seleccionadas.
- Puede ser con los nombres de las variables.
- Puede ser con variables lógicas

```
Log1=1:2<2;Log2=1:3>1
Mat1
```

```
##          Wood Watts Jones
## Jagger      1     3     5
## Richards    2     4     6
```

```
Mat1[Log1,Log2]
```

```
## Watts Jones
##     3     5
```

Subconjuntos

Hay que hacer mención especial a las listas. Ofrecen dos formas nuevas de subconjuntos:

- El doble corchete permite acceder al tipo de dato del elemento:

```
Lis1=list(Moe="a",Larry=1,Curly=TRUE)
Lis1[1]
```

```
## $Moe
## [1] "a"
```

```
Lis1[[1]]
```

```
## [1] "a"
```

Subconjuntos

Hay que hacer mención especial a las listas. Ofrecen dos formas nuevas de subconjuntos:

- El doble corchete permite acceder al tipo de dato del elemento:

```
Lis1=list(Moe="a",Larry=1,Curly=TRUE)
class(Lis1[1])
```

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

```
class(Lis1[[1]])
```

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

Subconjuntos

Hay que hacer mención especial a las listas. Ofrecen formas nuevas de subconjuntos:

- El doble corchete permite acceder al tipo de dato del elemento.
- Cuando los elementos tienen nombres, permite hacer algo similar al doble corchete con el signo $

```
Lis1=list(Moe="a",Larry=1,Curly=TRUE)
Lis1[[3]]
```

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

```
Lis1$Curly
```

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

Subconjuntos

Además, con estas nuevas variantes, permite agregar vectores, matrices y hasta otras listas a listas previas, apelando a nombres o elementos aún indefinidos

```
Lis1[[4]]=vec1
names(Lis1)[4]="Shemp"
Lis1
```

```
## $Moe
## [1] "a"
##
## $Larry
## [1] 1
##
## $Curly
## [1] TRUE
##
## $Shemp
##   Bart   Lisa Maggie
##      1      4      8
```

Subconjuntos

Además, con estas nuevas variantes, permite agregar vectores, matrices y hasta otras listas a listas previas, apelando a nombres o elementos aún indefinidos

```
Lis1$Joe=Lis1
Lis1[[5]]
```

```
## $Moe
## [1] "a"
##
## $Larry
## [1] 1
##
## $Curly
## [1] TRUE
##
## $Shemp
##   Bart   Lisa Maggie
##      1      4      8
```

Subconjuntos

Una última forma interesante de tomar subconjuntos es con el comando sample:

- Con reposición

```
dado=1:6
DosTiradas=sample(dado,2,replace = T)
DosTiradas
```

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

```
SeisTiradas=sample(dado,6,replace = T)
SeisTiradas
```

```
## [1] 2 5 1 6 5 1
```

Subconjuntos

Una última forma interesante de tomar subconjuntos es con el comando sample:

- Con reposición
- Sin reposición

```
dado=1:6
SeisTiradas=sample(dado,6,replace = F)
SeisTiradas
```

```
## [1] 5 6 1 2 4 3
```

Notar que en este caso termina siendo una permutación del vector 1:6

Subconjuntos

Una última forma interesante de tomar subconjuntos es con el comando sample:

- Con reposición
- Sin reposición
- Puede ser de cualquier tipo de vectores

```
Muestra=sample(c("A","B","C"),10,replace = T)
Muestra
```

```
##  [1] "C" "C" "C" "B" "C" "A" "A" "C" "C" "C"
```

```
Muestra=sample(c("A","B","C"),10,replace = F)
```

```
## Error in sample.int(length(x), size, replace, prob): cannot take a sample
```

Factores

Los factores son una forma de enfatizar que una variable es categórica. Se llaman niveles a los distintos valores que puede tomar esta variable.

Ejemplos:

```
Encuesta=sample(c("Sí","No"),10,replace = T)
Encuesta
```

```
##  [1] "No" "Sí" "Sí" "No" "Sí" "Sí" "Sí" "No" "Sí" "No"
```

```
class(Encuesta)
```

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

Factores

Los factores son una forma de enfatizar que una variable es categórica. Se llaman niveles a los distintos valores que puede tomar esta variable.

Ejemplos:

```
Encuesta=sample(c("Sí","No"),10,replace = T)
Fact.Encuesta=factor(Encuesta)
Fact.Encuesta
```

```
##  [1] Sí Sí Sí No No Sí Sí Sí Sí No
## Levels: No Sí
```

```
class(Fact.Encuesta)
```

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

Factores

Los factores son una forma de enfatizar que una variable es categórica. Se llaman niveles a los distintos valores que puede tomar esta variable.

Ejemplos:

```
Educacion=sample(c("Prim","Secu","Univ","PostG"),10,replace = T)
Fact.Educacion=factor(Educacion,
                      levels=c("Prim","Secu","Univ","PostG"),
                      ordered = T)
Fact.Educacion
```

```
##  [1] Univ  PostG PostG PostG Univ  Prim  Univ  Prim  Univ  Univ
## Levels: Prim < Secu < Univ < PostG
```

Factores

Los factores son muy útiles ya que R se encarga de que en el caso de tener que hacer una cuenta numérica, transforma los factores según sus niveles para evitar errores en la asignación numérica.

Sin embargo, los factores tienen muchos problemas con la coerción, conviene siempre pasarlas a caracteres:

```
Nums=sample(-1:1,10,replace = T)
Fact.Nums=factor(Nums)
Fact.Nums
```

```
##  [1] -1 1  1  1  1  0  1  1  1  0
## Levels: -1 0 1
```

```
as.numeric(Fact.Nums)
```

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

Factores

Los factores son muy útiles ya que R se encarga de que en el caso de tener que hacer una cuenta numérica, transforma los factores según sus niveles para evitar errores en la asignación numérica.

Sin embargo, los factores tienen muchos problemas con la coerción, conviene siempre pasarlas a caracteres primero:

```
Nums=sample(-1:1,10,replace = T)
Fact.Nums=factor(Nums)
Fact.Nums
```

```
##  [1] 0  -1 -1 0  -1 0  1  -1 -1 1
## Levels: -1 0 1
```

```
as.character(Fact.Nums)
```

```
##  [1] "0"  "-1" "-1" "0"  "-1" "0"  "1"  "-1" "-1" "1"
```

Data Frames

El objeto que sin duda hace la diferencia en R es el data.frame.

Ejemplo:

```
head(mpg)
```

```
##   manufacturer model displ year cyl      trans drv cty hwy fl   class
## 1         audi    a4   1.8 1999   4   auto(l5)   f  18  29  p compact
## 2         audi    a4   1.8 1999   4 manual(m5)   f  21  29  p compact
## 3         audi    a4   2.0 2008   4 manual(m6)   f  20  31  p compact
## 4         audi    a4   2.0 2008   4   auto(av)   f  21  30  p compact
## 5         audi    a4   2.8 1999   6   auto(l5)   f  16  26  p compact
## 6         audi    a4   2.8 1999   6 manual(m5)   f  18  26  p compact
```

```
class(mpg)
```

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

Data Frames

El objeto que sin duda hace la diferencia en R es el data.frame.

Ejemplo:

```
str(mpg)
```

```
## 'data.frame':    234 obs. of  11 variables:
##  $ manufacturer: chr  "audi" "audi" "audi" "audi" ...
##  $ model       : chr  "a4" "a4" "a4" "a4" ...
##  $ displ       : num  1.8 1.8 2 2 2.8 2.8 3.1 1.8 1.8 2 ...
##  $ year        : int  1999 1999 2008 2008 1999 1999 2008 1999 1999 2008
##  $ cyl         : int  4 4 4 4 6 6 6 4 4 4 ...
##  $ trans       : chr  "auto(l5)" "manual(m5)" "manual(m6)" "auto(av)" ..
##  $ drv         : chr  "f" "f" "f" "f" ...
##  $ cty         : int  18 21 20 21 16 18 18 18 16 20 ...
##  $ hwy         : int  29 29 31 30 26 26 27 26 25 28 ...
##  $ fl          : chr  "p" "p" "p" "p" ...
##  $ class       : chr  "compact" "compact" "compact" "compact" ...
```

Data Frames

El data frame tiene características

- de las matrices: Tiene una cantidad de filas y columnas fijas. Permite las mismas formas de tomar subconjuntos.

```
nrow(mpg);ncol(mpg)
```

```
## [1] 234
```

```
## [1] 11
```

Data Frames

El data frame tiene características

- de las matrices: Tiene una cantidad de filas y columnas fijas. Permite las mismas formas de tomar subconjuntos.

```
names(mpg)
```

```
##  [1] "manufacturer" "model"        "displ"        "year"         "cyl"
##  [6] "trans"        "drv"          "cty"          "hwy"          "fl"
## [11] "class"
```

```
Noms.A=grepl("a",names(mpg));Noms.A
```

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

```
mpg[c(2,4:6),Noms.A]
```

```
##   manufacturer year      trans   class
## 2         audi 1999 manual(m5) compact
## 4         audi 2008   auto(av) compact
## 5         audi 1999   auto(l5) compact
## 6         audi 1999 manual(m5) compact
```

Data Frames

El data frame tiene características

- de las matrices: Tiene una cantidad de filas y columnas fijas. Permite las mismas formas de tomar subconjuntos.
- de las listas: Puede albergar distintos tipos de datos y cada columna es un elemento.

```
head(mpg$drv)
```

```
## [1] "f" "f" "f" "f" "f" "f"
```

```
class(mpg$drv)
```

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

```
class(mpg$hwy)
```

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

Data Frames

El data frame tiene características

- de las matrices: Tiene una cantidad de filas y columnas fijas. Permite las mismas formas de tomar subconjuntos.
- de las listas: Puede albergar distintos tipos de datos y cada columna es un elemento.
- propias: A diferencia de los otros tipos de datos, el data.frame requiere al menos nombres en sus columnas.

Data Frames

Vamos a construir nuestro propio data.frame, que consiste de un mazo de cartas

Data Frames

Vamos a construir nuestro propio data.frame, que consiste de un mazo de cartas , con dos variables,

Data Frames

Vamos a construir nuestro propio data.frame, que consiste de un mazo de cartas , con dos variables, el palo

Data Frames

Vamos a construir nuestro propio data.frame, que consiste de un mazo de cartas , con dos variables, el palo (caracteres)

Data Frames

Vamos a construir nuestro propio data.frame, que consiste de un mazo de cartas , con dos variables, el palo (caracteres) y el número

Data Frames

Vamos a construir nuestro propio data.frame, que consiste de un mazo de cartas , con dos variables, el palo (caracteres) y el número (lógica,

Data Frames

Vamos a construir nuestro propio data.frame, que consiste de un mazo de cartas , con dos variables, el palo (caracteres) y el número (lógica, ahre. Claramente numérica).

```
Mazo=data.frame(
  Palo=rep(c("Espada","Basto","Oro","Copa"),each=10),
  Numero=rep(c(1:7,10:12),4))
head(Mazo)
```

```
##     Palo Numero
## 1 Espada      1
## 2 Espada      2
## 3 Espada      3
## 4 Espada      4
## 5 Espada      5
## 6 Espada      6
```

Data Frames

Vamos a construir nuestro propio data.frame, que consiste de un mazo de cartas, con dos variables, el palo (caracteres) y el número (lógica, ahre. Claramente numérica).

```
Mazo=data.frame(
  Palo=rep(c("Espada","Basto","Oro","Copa"),each=10),
  Numero=rep(c(1:7,10:12),4))
tail(Mazo)
```

```
##    Palo Numero
## 35 Copa      5
## 36 Copa      6
## 37 Copa      7
## 38 Copa     10
## 39 Copa     11
## 40 Copa     12
```

Data Frames

Vamos a construir nuestro propio data.frame, que consiste de un mazo de cartas, con dos variables, el palo (caracteres) y el número (lógica, ahre. Claramente numérica).

```
Mazo=data.frame(
  Palo=rep(c("Espada","Basto","Oro","Copa"),each=10),
  Numero=rep(c(1:7,10:12),4))
str(Mazo)
```

```
## 'data.frame':    40 obs. of  2 variables:
##  $ Palo  : Factor w/ 4 levels "Basto","Copa",..: 3 3 3 3 3 3 3 3 3 3 ...
##  $ Numero: int  1 2 3 4 5 6 7 10 11 12 ...
```

Funciones

Definiendo funciones

Para definir una función, ponemos primero el nombre de la función, el comando function luego los argumentos entre paréntesis. El cuerpo de la función va entre llaves.

```
Suma1=function(x){
  x=x+1
  return(x)
}
Suma1(3)
```

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

```
Suma1(TRUE)
```

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

```
Suma1("a")
```

```
## Error in x + 1: non-numeric argument to binary operator
```

Definiendo funciones

Para definir una función, ponemos primero el nombre de la función, el comando function luego los argumentos entre paréntesis. El cuerpo de la función va entre llaves.

```
Suma1=function(x){
  x=x+1
  return(x)
}
vec1
```

```
##   Bart   Lisa Maggie
##      1      4      8
```

```
Suma1(vec1)
```

```
##   Bart   Lisa Maggie
##      2      5      9
```

Definiendo funciones

Para definir una función, ponemos primero el nombre de la función, el comando function luego los argumentos entre paréntesis. El cuerpo de la función va entre llaves.

```
Suma1=function(x){
  x=x+1
  return(x)
}
Mat1
```

```
##           Wood Watts Jones
## Jagger       1     3     5
## Richards     2     4     6
```

```
Suma1(Mat1)
```

```
##           Wood Watts Jones
## Jagger       2     4     6
## Richards     3     5     7
```

Definiendo funciones

Para definir una función, ponemos primero el nombre de la función, el comando function luego los argumentos entre paréntesis. El cuerpo de la función va entre llaves.

```
Suma1=function(x){
  x=x+1
  return(x)
}
head(Suma1(Mazo))
```

```
## Warning in Ops.factor(left, right): '+' not meaningful for factors
```

```
##   Palo Numero
## 1   NA      2
## 2   NA      3
## 3   NA      4
## 4   NA      5
## 5   NA      6
## 6   NA      7
```

Definiendo funciones

Si dejamos una línea de comando sin asignación, el prompt no va a mostrar el valor. Para que lo muestre explícitamente, hay que usar el comando print

```
Suma1=function(x){
  x
  x=x+1
  x
  return(x)
}
Suma1(TRUE)
```

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

Definiendo funciones

Si dejamos una línea de comando sin asignación, el prompt no va a mostrar el valor. Para que lo muestre explícitamente, hay que usar el comando print

```
Suma1=function(x){
  print("¿Está cansado de que no le sumen uno?")
  print("x Antes:")
  print(x)
  x=x+1
  print("x Después:")
  print(x)
  return(x)
}
Suma1(TRUE)
```

```
## [1] "¿Está cansado de que no le sumen uno?"
## [1] "x Antes:"
## [1] TRUE
## [1] "x Después:"
## [1] 2


## [1] 2
```

Definiendo funciones

Ya que tenemos un mazo, vamos a mezclarlo de forma aleatoria.

```
Mezcla=function(Mazo){
  N=nrow(Mazo)
  Perm=sample(1:N,N,replace = F)
  Mazo.Mezc=Mazo[Perm,]
  return(Mazo.Mezc)
}
```

Definiendo funciones

Ya que tenemos un mazo, vamos a mezclarlo de forma aleatoria.

```
head(Mazo)
```

```
##     Palo Numero
## 1 Espada      1
## 2 Espada      2
## 3 Espada      3
## 4 Espada      4
## 5 Espada      5
## 6 Espada      6
```

```
Mezc.Mazo=Mezcla(Mazo)
head(Mezc.Mazo)
```

```
##     Palo Numero
## 26   Oro      6
## 15 Basto      5
## 39  Copa     11
## 40  Copa     12
## 32  Copa      2
## 34  Copa      4
```

Definiendo funciones

Ya que tenemos un mazo, vamos a repartir las cartas para K jugadores. Un truco (obvio) y 3 cartas para cada uno.

```
Repartir=function(Mazo,K){
  N=nrow(Mazo)
  Mano=list()
  if(N>=K*3){
    for(i in 1:K){
      Act.Jug=paste0("Jugador",i)
      Act.Mano=Mazo[i+(0:2)*K,]
      rownames(Act.Mano)=NULL
      Mano[[Act.Jug]]=Act.Mano
    }
  }else{
    stop("No alcanzan las cartas")
  }
  return(Mano)
}
```

Definiendo funciones

Ya que tenemos un mazo, vamos a repartir las cartas para K jugadores. Un truco (obvio) y 3 cartas para cada uno.

```
Repartir(Mazo,2)
```

```
## $Jugador1
##     Palo Numero
## 1 Espada      1
## 2 Espada      3
## 3 Espada      5
##
## $Jugador2
##     Palo Numero
## 1 Espada      2
## 2 Espada      4
## 3 Espada      6
```

Definiendo funciones

Ya que tenemos un mazo, vamos a repartir las cartas para K jugadores. Un truco (obvio) y 3 cartas para cada uno.

```
Repartir(Mazo,15)
```

```
## Error in Repartir(Mazo, 15): No alcanzan las cartas
```

Notar que cuando sumamos muchos jugadores la función tira un error.

Definiendo funciones

Ya que tenemos un mazo, vamos a jugar al truco.

```
Truco=function(Mazo,K){
  Mez=Mezcla(Mazo)
  Rep=Repartir(Mez,K)
  return(Rep)
}
Truco(Mazo,2)
```

```
## $Jugador1
##     Palo Numero
## 1  Basto      1
## 2   Copa      2
## 3 Espada      1
##
## $Jugador2
##    Palo Numero
## 1 Basto      4
## 2 Basto      3
## 3  Copa      7
```

Definiendo funciones

Para agregar valores de default a una función, se le agrega un igual dentro del paréntesis:

```
Mezcla=function(Mazo,Cartearse=F){
  N=nrow(Mazo)
  if(Cartearse){
    Ancho=which(Mazo$Palo=="Espada" & Mazo$Numero==1)
    MazoSinAncho=(1:N)[-Ancho]
    Perm=sample(MazoSinAncho,N-1,replace = F)
    PermCart=c(Perm[1],Ancho,Perm[-1])
    Mazo.Mezc=Mazo[PermCart,]
  }else{
    Perm=sample(1:N,N,replace = F)
    Mazo.Mezc=Mazo[Perm,]
  }
  return(Mazo.Mezc)
}
```

Definiendo funciones

Volvemos a definir la mano de truco con la posibilidad de cartearse.

```
Truco=function(Mazo,K,Cart=F){
  Mez=Mezcla(Mazo,Cart)
  Rep=Repartir(Mez,K)
  return(Rep)
}
Truco(Mazo,2,T)
```

```
## $Jugador1
##     Palo Numero
## 1   Copa      7
## 2 Espada     11
## 3 Espada      2
##
## $Jugador2
##     Palo Numero
## 1 Espada      1
## 2    Oro      7
## 3    Oro      6
```

Definiendo funciones

Volvemos a definir la mano de truco con la posibilidad de cartearse.

```
Truco=function(Mazo,K,Cart=F){
  Mez=Mezcla(Mazo,Cart)
  Rep=Repartir(Mez,K)
  return(Rep)
}
Truco(Mazo,2,T)
```

```
## $Jugador1
##    Palo Numero
## 1 Basto      1
## 2  Copa      5
## 3  Copa      3
##
## $Jugador2
##     Palo Numero
## 1 Espada      1
## 2  Basto     10
## 3    Oro     10
```

Definiendo funciones

Volvemos a definir la mano de truco con la posibilidad de cartearse.

```
Truco=function(Mazo,K,Cart=F){
  Mez=Mezcla(Mazo,Cart)
  Rep=Repartir(Mez,K)
  return(Rep)
}
Truco(Mazo,2)
```

```
## $Jugador1
##     Palo Numero
## 1  Basto      7
## 2 Espada      6
## 3    Oro     12
##
## $Jugador2
##    Palo Numero
## 1   Oro      3
## 2  Copa      6
## 3 Basto      2
```

Funciones aplicadas a vectores y listas

Una función muy utilizada en R también es lapply, en el que se aplica una función a una lista. En este caso, vamos a aplicar la función "Mezcla" a cada mano de la mano de truco que tenemos:

```
Mano.Act=Truco(Mazo,2)
Mano.Act
```

```
## $Jugador1
##     Palo Numero
## 1 Espada      3
## 2  Basto      6
## 3   Copa      3
##
## $Jugador2
##    Palo Numero
## 1 Basto     11
## 2  Copa      4
## 3 Basto     12
```

Funciones aplicadas a vectores y listas

Una función muy utilizada en R también es lapply, en el que se aplica una función a una lista. En este caso, vamos a aplicar la función "Mezcla" a cada mano de la mano de truco que tenemos:

```
lapply(Mano.Act,FUN = Mezcla)
```

```
## $Jugador1
##     Palo Numero
## 3   Copa      3
## 1 Espada      3
## 2  Basto      6
##
## $Jugador2
##    Palo Numero
## 2  Copa      4
## 3 Basto     12
## 1 Basto     11
```

Funciones aplicadas a vectores y listas

Esta función se usa mucho para data.frames, coordinado con una función llamada split, que divide una estructura de datos en una lista según un vector discreto (generalmente factores)

```
Split.MPG=split(mpg,mpg$drv)
lapply(Split.MPG, function(x){head(x[,-c(1,11)],n=2)})
```

```
## $`4`
##         model displ year cyl      trans drv cty hwy fl
## 8 a4 quattro   1.8 1999   4 manual(m5)   4  18  26  p
## 9 a4 quattro   1.8 1999   4   auto(l5)   4  16  25  p
##
## $f
##   model displ year cyl      trans drv cty hwy fl
## 1    a4   1.8 1999   4   auto(l5)   f  18  29  p
## 2    a4   1.8 1999   4 manual(m5)   f  21  29  p
##
## $r
##                  model displ year cyl    trans drv cty hwy fl
## 19 c1500 suburban 2wd   5.3 2008   8 auto(l4)   r  14  20  r
## 20 c1500 suburban 2wd   5.3 2008   8 auto(l4)   r  11  15  e
```

Funciones aplicadas a vectores y listas

Podemos usar esta estrategia, por ejemplo, para hacer una media de una cierta variable por cada nivel del factor

```
Split.MPG=split(mpg,mpg$drv)
lapply(Split.MPG, function(x){mean(x$hwy)})
```

```
## $`4`
## [1] 19.17476
##
## $f
## [1] 28.16038
##
## $r
## [1] 21
```

Funciones aplicadas a vectores y listas

El comando sapply hace lo mismo pero devolviendo un vector, que puede resultar un poco más amigable para instrucciones posteriores.

```
Split.MPG=split(mpg,mpg$drv)
sapply(Split.MPG, function(x){mean(x$hwy)})
```

```
##        4        f        r
## 19.17476 28.16038 21.00000
```