Introducción a R

Sesión 1: Panorama general de R

Jorge de la Vega Góngora

Maestría de Mercadotecnia,
Instituto Tecnológico Autónomo de México

30 de septiembre de 2021.

ITAM

Introducción

Antecedentes: un poco de historia

- El lenguaje S se desarrolló en Laboratorios AT & T–Bell, principalmente por John M. Chambers, en 1976. Participan también Rick Becker y Allan Wilks.
- Su uso se expandió rápidamente después de la publicación del libro de John Tukey: “Exploratory Data Analysis” (EDA).
- `S-Plus` fue una versión comercial de S que inició en 1987. Su popularidad incrementó dramáticamente después de 1990, y se mantuvo hasta la versión 8 en 2007.
- `S-Plus` fue atractivo porque contaba con una interfaz de usuario gráfica (GUI), y soportaba muchos formatos para importación y exportación de datos y gráficas.

Nacimiento de R I

- R se comenzó su desarrollo en los 90's, realizado por estadísticos como una alternativa de código abierto a S-Plus, en parte porque entonces no había una versión de S-Plus para Linux.
- Robert Gentleman y Ross Ihaka, de Nueva Zelanda, son los creadores pioneros de R.

- R se basa en el mismo lenguaje S utilizado en la versión 2000 de S-Plus, con algunas excepciones menores. Sin embargo, su arquitectura es un poco diferente.
- Ventajas: bien documentado; fácil de obtener e instalar; fácil de actualizar sus bibliotecas de funciones.
- Algunas desventajas de aquella época:
  - GUI muy básica (hasta la aparición de RStudio)

Nacimiento de R II

- manejo de memoria RAM ineficiente
- menos capacidades para importar y exportar datos de otros formatos
- no mucha interacción con Office

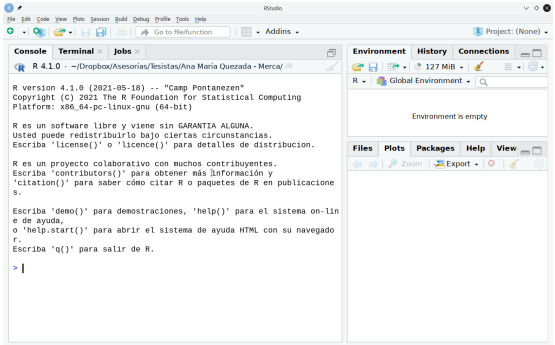
Estas desventajas se han ido resolviendo con el tiempo.

RStudio I

RStudio es una herramienta que facilita el trabajo con R, sobre todo en los siguientes puntos:

- Trabaja con R y sus gráficas de manera interactiva
- Permite organizar sus programas y tener organizados varios proyectos
- Permite realizar *investigación reproducible*, que esta tomando relevancia en el mundo científico
- Da mantenimiento a los paquetes instalados en el sistema
- Permite crear y compartir reportes
- Permite el intercambio de programas y la colaboración con otros usuarios
- Se puede llevar un control de proyectos en la nube conectando con github

Es tanto una interfaz gráfica (GUI) como un ambiente de desarrollo integrado (IDE). Hay versiones para Windows, Linux y Mac OS X. También permite correr R desde un servidor remoto.



¿Porqué se promueve el uso de R? I

- Existen muchos paquetes estadísticos orientados a procedimientos estadísticos: SAS, SPSS, Stata. Pero estos paquetes
  - Carecen de buenas gráficas interactivas
  - Es difícil implementar nuevos métodos
  - Algunos cálculos son cajas negras
  - Son excesivamente costosos
- R es gratuito con un lenguaje poderoso orientado a objetos.
- Permite análisis de datos y gráficas en forma interactiva y reproducible
- Es fácil implementar nuevos métodos y distribuir a otros usuarios
- Su programación y código son abiertos: siempre puedes saber qué estas haciendo
- Investigadores de punta en estadística son programadores de R y cuenta con una muy amplia y extensa comunidad a nivel global
- Millones de recursos (libros, tutoriales, videos, papers, etc.) disponibles de manera gratuita y de fácil acceso. Un Congreso anual de usuarios UseR!.
- R fue planeado para ser extensible

¿Porqué se promueve el uso de R? II

- Usuarios escriben nuevas funciones, al igual que los desarrolladores.
- La documentación para agregar funciones es excelente.
- Las funciones creadas por los usuarios se invocan igual que las internas.
- Usuarios pueden crear sus propios tipos de datos y agregar atributos, p.ej. comentarios a cada pieza de datos de R.

- R permite trabajar tanto con datos estructurados como no estructurados
- Es un ambiente para análisis estadístico y también un lenguaje de alto nivel: unos cuantos comandos hacen mucho trabajo.
- R permite crear las mejores gráficas científicas.
- En R ya hay varios paquetes, contribuidos por diferentes personas, que permiten aplicar eficiente y correctamente muchos métodos estadísticos.
- Permite concentrarse más en la interpretación de los resultados que en la implementación.

¿Porqué NO R?

No todo es maravilloso...

- Para utilizar R a su máximo potencial, la curva de aprendizaje es algo elevada.
- Se requiere, eventualmente, saber programar.
- El uso de paquetes está condicionado al menos parcialmente a los caprichos de su creador: puede decidir cuando cambiar cosas, dejar de dar mantenimiento, etc.
- Es posible que en el software/paquetes haya errores aunque usualmente se detectan, pero no siempre.
- Muchas empresas no permiten el uso de *Open Source* para sus procesos. Prefieren pagar por "soporte", (aunque en la práctica no sirva de nada).
- Es lento para algunas aplicaciones por su estructura, aunque hay maneras de hacerlo más eficiente.

Pasos a seguir

1. Primero hay que instalar R[1].
2. Instalar RStudio
3. Instalamos los paquetes complementarios que necesitemos para nuestro trabajo desde RStudio.
4. Creamos un proyecto de trabajo para cada uno de los temas que queramos trabajar en R. Cada proyecto debe tener su directorio específico.

[1] Para usuarios interesados en también usar LaTeX, el orden es: LaTeX $\rightarrow$ R $\rightarrow$ RStudio

Instalación de R

- Para Windows y Mac: Seguir las instrucciones indicadas en www.r-project.org[2].
- En cada sistema, puede que se requieran algunos programas específicos adicionales para desarrolladores si se van a compilar *paquetes* que usa R.
- Algunos paquetes de R pueden requerir también de algunas herramientas adicionales. Esto no es necesario resolver desde el principio, se puede resolver más adelante.
- Puede haber varias versiones de R instaladas en paralelo (cada una tiene su directorio).

[2] Alternativamente, se puede instalar una versión de R mejorada por Microsoft de https://mran.microsoft.com

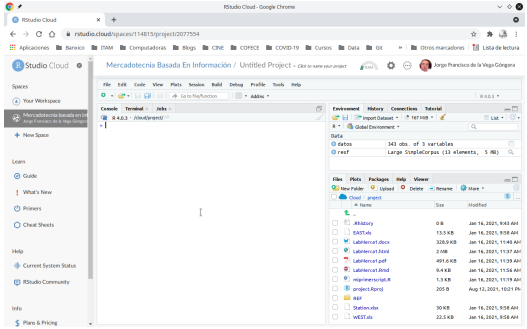
Instalación de paquetes

- Hay una gran cantidad de paquetes (*libraries*) que son colecciones de funciones empaquetadas que tienen un propósito definido. Los paquetes se publican en un repositorio conocido como CRAN (Comprehensive R Archive Network).
- Algunos paquetes en realidad son colecciones de otros paquetes, o bien algunos paquetes dependen de otros.
- Ejemplos de paquetes útiles que usaremos:
  - `knitr`
  - `rmarkdown`
  - `tidyverse`
  - `dplyr`
  - `forecast`
- Los paquetes se pueden instalar fácilmente desde `RStudio` a través de menús, o bien directamente desde la consola de `R`:

```
install.packages(c('knitr', 'rmarkdown','tydiverse','dplyr','forecast'))
```

Usando R y RStudio en linea

- Es posible usar una versión de RStudio en línea, a través de https://rstudio.cloud/.

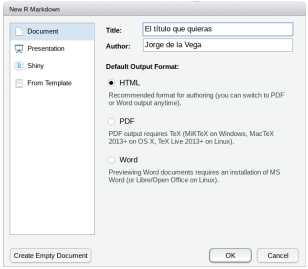


- Se puede usar una versión limitada en memoria y espacio de manera gratuita. Para iniciar el aprendizaje puede ser suficiente, pero en proyectos grandes se requiere más espacio.
- La ventaja es que se puede usar sin las complicaciones de instalación y/o configuración inicial. Usualmente todo fluye sin problemas.

Primeros pasos

Mi primer documento I

- Abrir `RStudio`
- `File` → `New File` → `R Markdown...`
- En la ventana que aparece, Poner título y dar click en OK



- Esto crea una plantilla en `markdown`[3]. Guardar con `Save as...` y poner un nombre de archivo con la extensión `.Rmd`

[3] Markdown es un lenguaje parecido a html pero muchísimo más simple que permite dar formatos marcando el texto y luego usando un intérprete para cambiar el formato

Formas de utilizar R

R se puede utilizar de varias maneras, dependiendo de la experiencia, conocimiento y necesidad de sistematización:

Consola De manera directa, a través de la consola de comandos

Gui A través de una interfaz de usuarios gráfica (GUI). En este tema, hay varias opciones, entre las más importantes están `RStudio`, `RCommander`, `Rattle`.

Batch En modo *Batch*, a través de un ícono.

Reglas generales para introducir comandos en R

- Comandos pueden introducirse en más de una linea. el prompt que indica la continuación de una linea es el signo más: +
- Comandos múltiples pueden ser introducidos en una misma linea separados por punto y coma (;)
- Los comentarios inician con #.
- Espacios y tabuladores son ignorados excepto cuando están entre comillas.
- R hace distinción entre mayúsculas y minúsculas.
- Se pueden usar las teclas ↑ o ↓ para navegar entre los comandos que han sido tecleados previamente.
- Se puede obtener ayuda de una función, por ejemplo `sin`, usando la ayuda HTML o bien tecleando `?sin` o `help(sin)`.
- La combinación de teclas `Ctrl + L` limpia la consola.

Ejemplos I

Los comandos más elementales consisten de expresiones o asignaciones.

```
2 + 3
```

```
[1] 5
```

```
sqrt(3/4)/(1/3-2*pi^2)
```

```
[1] -0.04462697
```

```
# genera datos aleatorios con distribución normal.
x <- rnorm(100,sd=4); y <- runif(100,-2,5)
# Los siguientes comandos devuelven por default los primeros y últimos 6 datos respectivamente
head(x); tail(x)
```

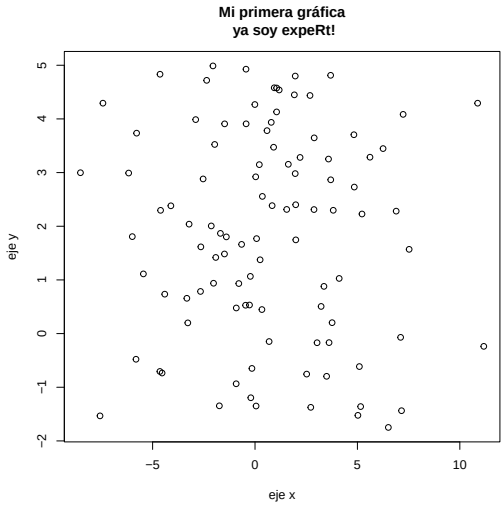
```
[1]  5.0235623 -0.9207391  1.9625725  1.9202891  6.5101368  2.8891659
[1] -2.0244635  1.9642245  7.2316267  0.0516095  3.6149959  1.5474515
```

```
head(x,10)  # Se puede cambiar el número de datos que queremos ver.
```

```
 [1]  5.0235623 -0.9207391  1.9625725  1.9202891  6.5101368  2.8891659
 [7] -4.5375758 -0.2056237 -4.6447640 -2.3603034
```

```
plot(x,y, main ="Mi primera gráfica\n ya soy expeRt!", xlab = "eje x", ylab = "eje y")
```

Ejemplos II



Ejemplos I

```r
z <- cbind(x,y) # 'pega' dos vectores por columnas
head(z)
```

```
              x          y
[1,]  5.0235623 -1.5235413
[2,] -0.9207391 -0.9351865
[3,]  1.9625725  2.9810946
[4,]  1.9202891  4.4486838
[5,]  6.5101368 -1.7487170
[6,]  2.8891659  3.6469373
```

```r
t(z) %*% z  # producto de matrices
```

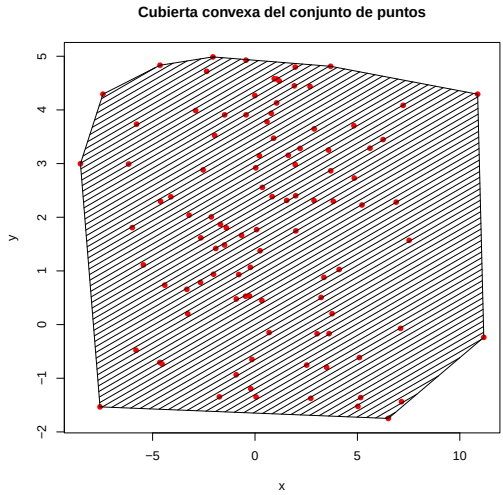
```
           x         y
x 1562.72561  73.10639
y   73.10639 721.80562
```

```r
crossprod(z)
```

```
           x         y
x 1562.72561  73.10639
y   73.10639 721.80562
```

```r
h <- chull(x,y)  # cubierta conveza del conjunto de puntos
plot(x, y, pch = 16, col = "red", main = "Cubierta convexa del conjunto de puntos")
polygon(x[h], y[h], dens = 15, angle = 30)
```

Ejemplos II



Tipos de objetos en R

Descripción concisa de objetos en R

En esta sección se describirán los principales tipos de objetos que se usan comunmente en R. Estos son:

- Vectores
- Matrices
- Arreglos
- Listas
- Factores
- Dataframes
- Funciones
- Indexación

A continuación veremos las características de cada uno de estos tipos de objetos.

Vectores I

Un vector es un conjunto *ordenado* de datos que son **del mismo tipo base o clase** (no se pueden mezclar). Cada elemento de un vector se le llama *componente*. Las clases pueden ser:

- numéricos `numeric`. Estos pueden ser a su vez de subclases: `integer` o `double` o `complex`

```
(x <- c(2,4,6))
[1] 2 4 6
class(x)
[1] "numeric"
1:5   #El símbolo ":" es para indicar una sucesión
[1] 1 2 3 4 5
```

- lógicos `logical`

Vectores II

```
(l <- c(TRUE,FALSE,TRUE,TRUE,FALSE))
[1]  TRUE FALSE  TRUE  TRUE FALSE
class(l)
[1] "logical"
class(l) <- "integer" #Podemos cambiar la clase para representar los valores lógicos como 0's y 1's
l
[1] 1 0 1 1 0
class(l) <- "logical" # y podemos regresarlos a su clase original.
l
[1]  TRUE FALSE  TRUE  TRUE FALSE
```

- caractéres `character`

```
(a <- c("a","b",'c'))  #Se pueden usar comillas dobles o sencillas, pero tienen que ser consistentes.
[1] "a" "b" "c"
(b <- c('a','b',"c"))
[1] "a" "b" "c"
class(b)
[1] "character"
```

Propiedades de los vectores I

Consideremos algunas propiedades y operaciones con los vectores.

- Cada componente de un vector se le puede asignar un nombre. El nombre es un *atributo* del vector.

```
x <- 1:5
colores <- c("rojo","amarillo",'negro','azul',"blanco")
names(x) <- colores  # modifica el atributo de nombres
x
```

```
    rojo amarillo    negro     azul   blanco
       1        2        3        4        5
```

```
attr(x,"names") #revisa cuál es el atributo de nombres
```

```
[1] "rojo"     "amarillo" "negro"    "azul"     "blanco"
```

- Otro atributo del vector es su longitud (`length`): es decir, el número de elementos que tiene el vector

Propiedades de los vectores II

```
length(x)
```

```
[1] 5
```

```
length(x) <- 8  #¿Qué pasa si cambiamos la longitud del vector a una mayor?
x
```

```
    rojo amarillo    negro     azul   blanco
       1        2        3        4        5       NA       NA       NA
```

```
length(x) <- 3 # ¿a una menor longitud?
x
```

```
    rojo amarillo    negro
       1        2        3
```

```
length(x) <- 5 # se pierde información si lo regresamos a su tamaño? SI
x
```

```
    rojo amarillo    negro
       1        2        3       NA       NA
```

Propiedades de los vectores III

- Los vectores se pueden combinar. Cuando los vectores no son del mismo tipo, R cambia el tipo base al más general posible: esto se llama *coerción*. Esto es necesario para mantener el mismo tipo de datos.

```
x <- 1:5; y <- c("letras","no números")
z <- c(x,y) # se combinan los vectores con la función de concatenación
z
[1] "1"          "2"          "3"          "4"          "5"
[6] "letras"     "no números"
```

- Uso de índices para accesar componentes del vector

```
x[2]   #podemos tomar un elemento
[1] 2
x[1:3] # o un subconjunto del vector.
[1] 1 2 3
x[10] # cuando se usa un índice que no es parte del vector
[1] NA
x[c(1,1,3)]  #los índices se pueden duplicar
[1] 1 1 3
```

o por nombre si lo tiene:

Propiedades de los vectores IV

```
x["rojo"]
[1] NA
```

Para quitar elementos usamos un signo negativo:

```
x[-1]
[1] 2 3 4 5
x[-c(1,3)]  # quita más de uno en diferentes posiciones.
[1] 2 4 5
x           # notar que no se modifica el vector con las operaciones anteriores, porque no están asignadas, sólo ca
[1] 1 2 3 4 5
```

- Otras formas de crear vectores:

Propiedades de los vectores V

```
z <- numeric(10)  # crea un vector de longitud 10, numérico. No tiene valores.
z
 [1] 0 0 0 0 0 0 0 0 0 0
z <- character(10)
z
 [1] "" "" "" "" "" "" "" "" "" ""
z <- logical(10)
z
 [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
z <- numeric(0)  # define un vector sin elementos (vacío) es mejor que z <- NULL. Es mejor definir los objetos del t
z
numeric(0)
```

Aritmética con vectores I

- Las operaciones aritméticas de los vectores se llevan componente a componente:

```
x <- c(1,4,5,2,10)
y <- c(3,1,2,5,6)
x + y
```
```
[1]  4  5  7  7 16
```
```
x - y
```
```
[1] -2  3  3 -3  4
```
```
x * y
```
```
[1]  3  4 10 10 60
```
```
x/y
```
```
[1] 0.3333333 4.0000000 2.5000000 0.4000000 1.6666667
```

- Si un vector se divide o multiplica por un número, cada componente del vector se multiplica/divide por ese número. Si un número se divide por un vector, se obtiene un vector el número dividido por cada componente

```
w <- 1:5
2*w
```
```
[1]  2  4  6  8 10
```
```
2/w
```
```
[1] 2.0000000 1.0000000 0.6666667 0.5000000 0.4000000
```

Aritmética con vectores II

- Regla del reciclaje: ¿Qué pasa si intentamos hacer operaciones con vectores que tienen diferente longitud? Resulta la siguiente regla:

El vector más corto se reciclará para que tenga el tamaño del vector más grande

```
u <- c(10,20,30)
v <- 1:9
u + v
```

```
[1] 11 22 33 14 25 36 17 28 39
```

Hay que tener esta regla presente y tener mucho cuidado al usarla o no darse cuenta de que se está aplicando.

Un tipo especial de vector: los *factores* I

Factores

- Los objetos llamados *factores* son un tipo especial de vector que típicamente se utilizan para guardar variables categóricas o etiquetas que sirven para *clasificación*. Estas variables tienen un significado especial en estadística, particularmente en el análisis de regresión y en el diseño de encuestas, como veremos más adelante en el diplomado.
- La característica importante de un factor es que etiquetan observaciones a pertenecer a cierto grupo o categoría. Por ejemplo, la religión, el sexo, el estado civil, el estado de la república, etc, son ejemplos de variables que pueden considerarse factores cuando se usan para separar grupos.
- Los *niveles* o etiquetas de los grupos no son relevante en sí, sólo el hecho de que distinguen a las observaciones en diferentes grupos. Se puede usar 'H' y 'M' o bien 0 y 1 o bien 1 y 0, para distinguir las observaciones de hombres y de mujeres.

Un tipo especial de vector: los *factores* II

```
# variable con los tipos de crédito: hip = hipotecario, TC = tarjeta de crédito, per = personal.
credito <- factor(c("hip","TC","per","per","hip","per","TC","TC"))
credito

[1] hip TC  per per hip per TC  TC
Levels: hip per TC
```

En un factor, R ordena Los niveles alfabéticamente. Algunas funciones le dan un valor especial al primer nivel, por lo que a veces se requiere dar explícitamente los niveles.
En ciertos casos, los grupos pueden tener tener un orden, por ejemplo, si se considera ingreso, se puede pensar en un ingreso bajo, medio o alto. En ese caso se dice que el factor es *ordinal*. Cuando los factores son ordinales, se puede indicar el orden deseado de las etiquetas:

```
ingreso <- ordered(c("M","B","M","A","M","B","B","A"), levels = c("B","M","A"))
ingreso

[1] M B M A M B B A
Levels: B < M < A
```

Un tipo especial de vector: los *factores* III

Una variable que es continua pero que se quiere separar en rangos para definir grupos (por ejemplo, el ingreso bajo puede ir de 0 a 1000 mensuales), se puede *discretizar* usando la función `cut`:

```
z <- rnorm(20)
z

 [1] -0.810980239  2.176708284  0.798925947 -2.056034845 -2.010258773
 [6]  1.615133308 -0.745595279 -0.905284318 -0.438380945  0.081215530
[11] -0.395133110  0.922603553 -0.200141157 -0.005925389 -1.959354317
[16]  0.526368630 -0.117987381 -1.401192944  0.014700109 -1.529253185

u <- cut(z,breaks = c(-4,-2,0,2,4))  #damos los puntos de corte
u <- ordered(u, levels(u))
u

 [1] (-2,0]  (2,4]   (0,2]   (-4,-2] (-4,-2] (0,2]   (-2,0]  (-2,0]  (-2,0]
[10] (0,2]   (-2,0]  (0,2]   (-2,0]  (-2,0]  (-2,0]  (0,2]   (-2,0]  (-2,0]
[19] (0,2]   (-2,0]
Levels: (-4,-2] < (-2,0] < (0,2] < (2,4]

levels(u) <- c("B","MB","MA","A") #Podemos reetiquetar los niveles para hacerlos más fáciles de leer o entender
```

Matrices

Una *matriz* es un conjunto de datos arreglados en un rectángulo en dos dimensiones (tiene renglones y columnas). Por ejemplo, la siguiente matriz tiene dos renglones y tres columnas:

$$\mathbf{A} = \begin{bmatrix} 2 & 4 & 3 \\ 1 & 5 & 7 \end{bmatrix}$$

También se puede pensar en una matriz como un vector con atributo de dimensión (`dim`). Entonces, de hecho, las matrices en `R` son también vectores.

Veamos unos ejemplos:

Matrices II

```
x <- 1:20
x
```

```
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
```

```
dim(x) <- c(2,10) #le asignamos dimensión al vector x
x
```

```
     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
[1,]    1    3    5    7    9   11   13   15   17    19
[2,]    2    4    6    8   10   12   14   16   18    20
```

```
dim(x) <- NULL  #¿qué pasa si le quitamos su dimensión? También se puede usar c(x)
x
```

```
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
```

Para crear una matiz, se puede usar la siguiente forma

Matrices III

```
A <- matrix(c(2,4,6,7,8,4,5,6,2,1), # los datos
            nrow = 2,               # el número de renglones
            ncol = 5)               # el número de columnas
A
```

```
     [,1] [,2] [,3] [,4] [,5]
[1,]    2    6    8    5    2
[2,]    4    7    4    6    1
```

La matriz también se puede llenar por renglones usando índices para sus componentes

```
A <- matrix(
     c(2,4,6,7,8,4,5,6,2,1),
     nrow = 2,
     ncol = 5,
     byrow = T)              # Se indica que se llene por renglón
A
```

```
     [,1] [,2] [,3] [,4] [,5]
[1,]    2    4    6    7    8
[2,]    4    5    6    2    1
```

Los elementos de la matriz se pueden accesar usando:

Matrices IV

```
A[2,3]  # El elemento en el renglón 2 y columna 3

[1] 6

A[2, ]  # Todo el renglón 2, notar que es un vector

[1] 4 5 6 2 1

A[ ,3]  # Toda la columna 3, notar que es un vector

[1] 6 6

A[,c(1,3)] # Columnas 1 y 3, es una sub-matriz

     [,1] [,2]
[1,]    2    6
[2,]    4    6
```

El número de renglones y columnas pueden obtenerse con las funciones `nrow` y `ncol`.

Matrices V

```
nrow(A)
```

```
[1] 2
```

```
ncol(A)
```

```
[1] 5
```

Podemos asignar nombres a los renglones y a las columnas de la matriz, y como en el caso de los vectores, podemos accesar a los elementos por los nombres:

```
dimnames(A) = list(                 # La función para definir los nombres de las dimensiones, tiene que ser una lista.
                c("r1", "r2"),      # nombres de renglones
                c("c1", "c2", "c3","c4","c5"))   # nombres de columnas

A
```

```
   c1 c2 c3 c4 c5
r1  2  4  6  7  8
r2  4  5  6  2  1
```

```
A["r2", "c3"] # elemento en el renglón 2 y columna 3
```

```
[1] 6
```

Operaciones con matrices I

La *transpuesta* de una matriz intercambia los renglones y columnas. La notación matemática de la transpuesta de una matriz A es A^T o A' o A^t.

```
# No es necesario dar el número de columnas cuando tenemos la longitud
# y el número de renglones:

B <- matrix(1:6,nrow=3)
B

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

t(B)      # transpuesta de B

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

Así como con los vectores, si multiplicamos dos matrices que tengan las mismas dimensiones, la multiplicación se hace entrada a entrada:

Operaciones con matrices II

```
B*B

     [,1] [,2]
[1,]    1   16
[2,]    4   25
[3,]    9   36
```

Pero ese tipo de producto no es el *producto de matrices* que se utiliza en matemáticas. El producto matricial tiene una notación especial: %*%s y sólo se puede hacer si el número de renglones de una matriz es igual al número de columnas de la otra matriz, es decir, una matriz es de dimensiones $r \times c$ y la otra debe ser $c \times q$.

```
B %*% t(B)

     [,1] [,2] [,3]
[1,]   17   22   27
[2,]   22   29   36
[3,]   27   36   45
```

Podemos *combinar* matrices si tienen el mismo número de renglones, o el mismo número de columnas para crear una matriz más grande:

Operaciones con matrices III

```
C <- matrix(c(7,4,1), nrow = 3)  #C tiene dimensiones 3x1
# Combinamos B y C por columnas (porque ambas tienen el mismo número de renglones:
cbind(C,B)

     [,1] [,2] [,3]
[1,]    7    1    4
[2,]    4    2    5
[3,]    1    3    6

cbind(B,C)

     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    4
[3,]    3    6    1
```

Para combinar por renglones, usamos la función `rbind`

factores como matrices

En realidad, los objetos que llamamos factores, en realidad son matrices.

Arreglos I

Arreglos

Los *Arreglos* son una extensión de las matrices, en donde se pueden agregar más dimensiones (`dim`) de longitud 3 o más.

Por ejemplo:

```
x <- 1:30
dim(x) <- c(5,2,3)  # es como un 'cubo'
dimnames(x) <- list(color = colores, tipo = c("gordo", "flaco"), var = c("H", "M", "N"))
```

Los elementos se pueden accesar como en el caso de vectores.

Arreglos II

```
x[3,2,1]

[1] 8

x[,,1]   # caras del arreglo

          tipo
color      gordo flaco
  rojo         1     6
  amarillo     2     7
  negro        3     8
  azul         4     9
  blanco       5    10

x[2,,3]

gordo flaco
   22    27
```

Arreglos III

La convención en `R` para *llenar* un arreglo es la siguiente: el primer índice es el que se 'mueve' más rápido y el último es el que se 'mueve' más lento.
Un arreglo sigue siendo un vector, pero con más dimensiones. Acomodar las cosas en dimensiones puede ser útil para *organizar la información*.

Listas I

listas

Una *lista* es un objeto que sirve para contener otros objetos que pueden ser todos de diferentes tipos.

- Esta puede ser la estructura de datos más interesante para almacenar datos que se conocen como *no-estructurados*. El manejo de las listas puede ser un poco más difícil de entender, por la manera en que tenemos que accesar a sus elementos. Lo veremos a través de un ejemplo concreto
- Consideremos un crédito a una persona. El crédito tiene varias características importantes que se tienen que definir:
  - El tipo: puede ser hipotecario, tarjeta de crédito, personal, automotriz.
  - El monto del préstamo
  - La tasa de interés que tiene el crédito
  - Su plazo (en meses)
  - su fecha de contratación
  - datos demográficos de la persona que recibe el crédito (edad, sexo, estado civil).

Listas II

Podemos definir una lista con todas estas características:

```
Credito <- list(tipo = "Hipotecario",
                plazo = 28,
                fecha = "01/04/20",
                tasa = 7.58,
                demograficos = c(35,1,0))
Credito
```

```
$tipo
[1] "Hipotecario"

$plazo
[1] 28

$fecha
[1] "01/04/20"

$tasa
[1] 7.58

$demograficos
[1] 35  1  0
```

Listas III

- A los elementos de la lista se puede accesar ya sea por posición o por nombre:

```
   Credito[[3]] #tercer objeto en la lista
[1] "01/04/20"
   Credito[3]  # lista con un componente
$fecha
[1] "01/04/20"
   Credito$fecha  #componente con nombre fecha
[1] "01/04/20"
   Credito[3:4] #se seleccionan varios elementos como vector
$fecha
[1] "01/04/20"

$tasa
[1] 7.58
```

Se nos olvidó agregar el monto:

Listas IV

```
Credito <- c(Credito, monto = 1.2)  #se agrega el monto del crédito, en millones
Credito
```

```
$tipo
[1] "Hipotecario"

$plazo
[1] 28

$fecha
[1] "01/04/20"

$tasa
[1] 7.58

$demograficos
[1] 35  1  0

$monto
[1] 1.2
```

Elimina el componente monto de la lista

Listas V

```
Credito$monto <- NULL  #o Credito[["monto"]] <- NULL
Credito
```

```
$tipo
[1] "Hipotecario"

$plazo
[1] 28

$fecha
[1] "01/04/20"

$tasa
[1] 7.58

$demograficos
[1] 35  1  0
```

Listas VI

- La función `unlist` convierte una lista en un vector, pero se aplica coerción para que todos los objetos tengan el mismo tipo, y cuando un objeto tiene una dimensión mayor, agrega cada componente numerado secuencialmente.

```
unlist(Credito)
```

```
        tipo         plazo         fecha          tasa demograficos1
"Hipotecario"          "28"    "01/04/20"        "7.58"          "35"
demograficos2 demograficos3
          "1"           "0"
```

Dataframes I

- Un *dataframe* es una lista hecha de vectores de la misma longitud, pero pueden ser de diferente tipo. Es la estructura más adecuada para guardar datos estructurados y lo más cercano a lo que se conoce como una *base de datos*.
- Un dataframe tiene atributos que las listas no tienen. La función `data.frame` genera un dataframe. Se puede usar la función `cbind` o `rbind` como si los datos fueran una matriz, pero regresa un dataframe:

Dataframes II

```
z <- data.frame(inst = factor(c("Cete","Cete","M0","M1","M1")),
                precio = c(9.91,9.93,9.988,9.87,9.67),
                ven = rep("01/10/04",5))
 z
```

```
  inst precio      ven
1 Cete  9.910 01/10/04
2 Cete  9.930 01/10/04
3   M0  9.988 01/10/04
4   M1  9.870 01/10/04
5   M1  9.670 01/10/04
```

```
 (z2 <- cbind(z,z))
```

```
  inst precio      ven inst precio      ven
1 Cete  9.910 01/10/04 Cete  9.910 01/10/04
2 Cete  9.930 01/10/04 Cete  9.930 01/10/04
3   M0  9.988 01/10/04   M0  9.988 01/10/04
4   M1  9.870 01/10/04   M1  9.870 01/10/04
5   M1  9.670 01/10/04   M1  9.670 01/10/04
```

Dataframes III

```
attributes(z)
```

```
$names
[1] "inst"   "precio" "ven"

$class
[1] "data.frame"

$row.names
[1] 1 2 3 4 5
```

- Los `dataframes` pueden ser indexados como matrices o como listas.

Dataframes IV

```
z[2] #es un dataframe
```

```
  precio
1  9.910
2  9.930
3  9.988
4  9.870
5  9.670
```

```
z[[2]] #es un vector, que es el segundo elemento de la lista
```

```
[1] 9.910 9.930 9.988 9.870 9.670
```

```
z[z$inst == 'Cete',]  # selecciona sólo los instrumentos que son Cetes.
```

```
  inst precio      ven
1 Cete   9.91 01/10/04
2 Cete   9.93 01/10/04
```

Dataframes V

- Matrices y listas pueden ser anexadas a un dataframe:

```
y <- matrix(rnorm(10),nrow=5)
y
```

```
           [,1]       [,2]
[1,] -1.3485144  0.1379018
[2,] -0.7900931  0.8714081
[3,] -1.3540708 -1.3356176
[4,] -0.3269887  0.5251535
[5,] -0.8293440 -1.7170616
```

```
z <- data.frame(z,y)
z
```

```
  inst precio      ven          X1          X2
1 Cete  9.910 01/10/04 -1.3485144  0.1379018
2 Cete  9.930 01/10/04 -0.7900931  0.8714081
3   M0  9.988 01/10/04 -1.3540708 -1.3356176
4   M1  9.870 01/10/04 -0.3269887  0.5251535
5   M1  9.670 01/10/04 -0.8293440 -1.7170616
```

Indexación I

Qué es la indexación?

La indexación consiste en extraer partes de un objeto en R, típicamente de alguna de las estructuras de datos.

Para vectores, los vectores índices pueden ser de 5 tipos:

1. Una proposición lógica que se puede valuar como verdadera o falsa para cada elemento del vector:

```
y

           [,1]       [,2]
[1,] -1.3485144  0.1379018
[2,] -0.7900931  0.8714081
[3,] -1.3540708 -1.3356176
[4,] -0.3269887  0.5251535
[5,] -0.8293440 -1.7170616

y < 0

     [,1]  [,2]
[1,] TRUE FALSE
[2,] TRUE FALSE
[3,] TRUE  TRUE
[4,] TRUE FALSE
[5,] TRUE  TRUE

y[y < 0]

[1] -1.3485144 -0.7900931 -1.3540708 -0.3269887 -0.8293440 -1.3356176 -1.7170616
```

Indexación II

2. Un vector de enteros positivos o un factor:

```
z[z$inst == "MO", ]
```

```
  inst precio      ven        X1        X2
3   MO  9.988 01/10/04 -1.354071 -1.335618
```

3. Un vector de enteros negativos: sirve para *excluir* términos:

```
y[-c(2,5,8)]
```

```
[1] -1.3485144 -1.3540708 -0.3269887  0.1379018  0.8714081  0.5251535 -1.7170616
```

4. Un vector de cadenas de caracteres: esto sólo aplica cuando el objeto tiene nombres. De otra forma, devuelve `NA`.
5. La posición de índice es vacía. `y[]`. Se comporta como si se reemplazara el índice por `1:length(y)`.
6. Un arreglo puede ser indexado por una matriz: si el arreglo tiene k índices, la matriz índice debe ser de dimensiones $m \times k$ y cada renglón de la matriz es usado como un conjunto de índices especificando un elemento del arreglo.

Indexación III

- Índices ceros no caen en ningún caso anterior: un índice 0 en un vector ya creado pasa nada y un índice 0 en un vector al que se le asigna un valor no lo acepta:

```
a <- 1:4
a[0]

integer(0)

a[0] <- 10
a

[1] 1 2 3 4
```

Ejercicios

Ejercicio

- R tiene algunos archivos de datos precargados para mostrar cómo se realizan algunos análisis de datos. En este caso, carguen el conjunto de datos que se llama `state.x77` que contiene algunas estadísticas de cada estado de los EUA, para el año 1977:

```
head(state.x77)
```

	Population	Income	Illiteracy	Life Exp	Murder	HS Grad	Frost	Area
Alabama	3615	3624	2.1	69.05	15.1	41.3	20	50708
Alaska	365	6315	1.5	69.31	11.3	66.7	152	566432
Arizona	2212	4530	1.8	70.55	7.8	58.1	15	113417
Arkansas	2110	3378	1.9	70.66	10.1	39.9	65	51945
California	21198	5114	1.1	71.71	10.3	62.6	20	156361
Colorado	2541	4884	0.7	72.06	6.8	63.9	166	103766

- Con los datos anteriores, hacer lo siguiente:
  - Calculen la media de analfabetismo y de asesinatos de ese año.
  - Ordenar los valores de menor a mayor de acuerdo al número de asesinatos, usando la función `sort`
  - Hagan una gráfica que muestre los datos de manera útil.
  - Identifiquen los estados que tienen una tasa de asesinatos mayor al 12 %.
  - Grafica los datos para tasa de analfabetismo y número de asesinatos.
  - Obten el subconjunto de datos que empiezan con ”M”.

Lectura de datos desde archivos

Importando archivos de datos externos

- R puede leer casi cualquier formato de datos. En este curso nos concentraremos principalmente en archivos `.csv`
- La mejor manera de trabajar con archivos de Excel que tienen una estructura de `dataframe` es guardarlos como archivos csv y leerlos con la función `read.csv`.
- Otra función útil es `read.table`. De hecho, `read.csv` es lo que se conoce como un "wrapper" o envoltura de la función `read.table`, en el sentido de que la función original es muy cruda y la función envoltura la hace presentable al usuario, pero llama a la función original.
- Se pueden leer tanto archivos locales como archivos en línea. Por ejemplo:

```
# Archivos publicados por Sociedad Hipotecaria Federal
datos <- read.csv(file = "https://github.com/jvega68/Simulacion/raw/master/datos/SHF-PromedioValorMercadoxMetroCuadradoConstruccion.csv")
head(datos)
```

```
          Estado       Municipio    CP  vxm2
1 AGUASCALIENTES  AGUASCALIENTES   8320 12741
2 AGUASCALIENTES  AGUASCALIENTES  20000  4444
3 AGUASCALIENTES  AGUASCALIENTES  20010  7856
4 AGUASCALIENTES  AGUASCALIENTES  20016  7470
5 AGUASCALIENTES  AGUASCALIENTES  20020  7598
6 AGUASCALIENTES  AGUASCALIENTES  20029 10193
```

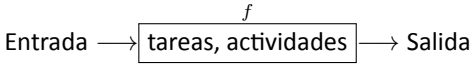
```
str(datos) # da una descripción general de los datos
```

```
'data.frame': 8076 obs. of  4 variables:
 $ Estado   : chr  "AGUASCALIENTES " "AGUASCALIENTES " "AGUASCALIENTES " "AGUASCALIENTES " ...
 $ Municipio: chr  "AGUASCALIENTES " "AGUASCALIENTES " "AGUASCALIENTES " "AGUASCALIENTES " ...
 $ CP       : int  8320 20000 20010 20016 20020 20029 20030 20050 20064 20070 ...
 $ vxm2     : int  12741 4444 7856 7470 7598 10193 5700 10100 7335 4898 ...
```

Funciones en R

Uso de funciones en R

- En R, la definición de funciones es muy similar al concepto que se utiliza en matemáticas. En un sentido práctico, una función toma una entrada, a esa entrada le hace algo, y devuelve una salida:



- Una de las fortalezas de R es su enfoque funcional; prácticamente todos los objetos son modificados por funciones,
- El usuario puede crear sus propias funciones. Los usuarios de R interactúan con el software principalmente a través de funciones.
- Ya hemos visto en las sesiones previas varios ejemplos de funciones. Las funciones se pueden agrupar en varios tipos:
  - funciones para transformar datos: matemáticas y estadísticas
  - funciones para leer, importar, manipular y exportar datos
  - funciones para graficar datos

Funciones de usuario I

- Las entradas de las funciones, se especifican a través de argumentos.
- La estructura básica de una función en R es:

```
        nombre.de.mi.funcion <- function(argumentos){
#poner aquí todos los pasos a seguir
}
```

- Las funciones pueden devolver como salida números, vectores, matrices, dataframes, listas, mensajes o gráficas.

Ejercicio

Crear una función, que tome un vector `x` y devuelva la raíz cuadrada de la suma de cada uno de sus componentes al cuadrado. Llama a la función `norma`.

Algunas propiedades de las funciones. I

- Si se llama al nombre de una función sin los paréntesis, en muchos casos se puede ver el código de la función. Cuando no se puede ver, es porque la función está *compilada*.
- Una función puede ser terminada usando los comandos `return`, `stop` o bien mandando un mensaje con el comando `warning`. En este último caso, la función continúa su evaluación.
- Una función se puede definir dentro de otra función. Si se define la función `f2` dentro de la función `f1`, entonces:
  - Si llamo a `f2` dentro de `f1`, se usará en forma anidada, y
  - La función `f2` no será visible fuera de `f1`

```
gx <- function(x){
              y2 <- function(y){ rnorm(1, mean = y, sd = y^2)}
        x^2 + y2(x+3)
        }
gx(10)
```

```
[1] -185.4186
```

```
y2(5)
```

```
Error in y2(5): no se pudo encontrar la función "y2"
```

Argumentos de funciones I

- Un tipo de argumento especial de las funciones son los tres puntos `....` Usualmente se usa para pasar posibles argumentos de una función a otra sin ponerlos explícitos
- También se usan cuando hay un número variable de argumentos.
- Las funciones pueden tener sus argumentos *especificados* o *no especificados* (. . .). Por ejemplo:

```
mi.funcion <- function(x, y = 1, ...){
********
}
```

En este ejemplo, los argumentos son `x`, que el usuario tiene que especificar, `y` que en caso de que el usuario no especifique tomará el valor de 1, y los tres puntos indican que puede haber más parámetros no especificados.

- Los argumentos *formales* son los que se usan en la definición de la función con un nombre, y los *reales* son los que se usan en una llamada a la función.

```
mi.funcion(4, color = T)
```

- Para conocer los argumentos de una función `f` usamos `args(f)`

```
args(sum)
```

```
function (..., na.rm = FALSE)
NULL
```

```
args(sqrt)
```

```
function (x)
NULL
```

Reglas de argumentos I

Hay una serie de reglas que definen cómo se deben aparejar los argumentos *formales* y los *reales*.

1. Los argumentos reales que se dan de la forma `nombre = valor` donde el nombre es *exactamente* el nombre de un argumento formal, son apareados primero. Si el argumento formal aparece después de . . ., ésta es la única forma en que será apareado.

```
        args(seq.int)
function (from, to, by, length.out, along.with, ...)
NULL
        seq.int(from = 10, to = 20, by = 2)
[1] 10 12 14 16 18 20
        seq.int(from = 10, to = 20, length.out = 5)
[1] 10.0 12.5 15.0 17.5 20.0
```

2. Si hay argumentos reales sin nombre, son apareados a los parámetros formales uno por uno en la sucesión de argumentos dados.

```
seq.int(10,20,2)
[1] 10 12 14 16 18 20
```

Reglas de argumentos II

3. Argumentos especificados de la forma `nombre = valor` para los que hay una correspondencia parcial con un argumento formal, son apareados.

```
seq.int(10,20,length.out=5)
[1] 10.0 12.5 15.0 17.5 20.0
```

4. Todos los restantes argumentos reales que no están apareados, formarán parte del argumento formal . . ., si hay uno, y si no se provee, entonces ocurre un error.

```
sqrt(10,20) #noten que es diferente a:
Error in sqrt(10, 20): 2 arguments passed to 'sqrt' which requires 1
sqrt(c(10,20))
[1] 3.162278 4.472136
```

5. Tener argumentos formales no apareados NO es un error, como se vió en el caso se `seq.int`

Funciones especiales para manipulación de datos

Uso de las funciones apply, lapply, tapply y sapply I

- La función apply permite aplicar otras funciones sobre los renglones o las columnas de una matriz.

```
args(apply)            # vemos sus argumentos

function (X, MARGIN, FUN, ..., simplify = TRUE)
NULL

     x <- runif(100)
     dim(x) <- c(25,4)      # crea una matriz de datos artificiales
     apply(x,2,mean)        # el MARGIN es 1 para renglones, 2 para columnas

[1] 0.4683151 0.5816197 0.4546884 0.5855156
```

- La función equivalente para listas es la función lapply, que aplica una función cada componente de una lista.

Uso de las funciones `apply`, `lapply`, `tapply` y `sapply` II

```
    y <-list(NULL)
    length(y) <- 2
    Y <- lapply(y,c,0)
    Z <- lapply(Y,function(x){runif(10)})
    Z[1:2]

[[1]]
 [1] 0.30088538 0.92522040 0.99683460 0.50293222 0.04207788 0.18256798
 [7] 0.99182926 0.24554736 0.66730185 0.29448298

[[2]]
 [1] 0.5594434 0.6563225 0.9972047 0.8395967 0.4881990 0.7062591 0.8064112
 [8] 0.6175566 0.9315812 0.5624030

    lapply(Z,max)  # Notar que se hace el cálculo, pero no se asigna a una variable.

[[1]]
[1] 0.9968346

[[2]]
[1] 0.9972047
```

- La función `tapply` aplica funciones sobre *grupos* de un dataframe o un array (identificados por una variable categórica); es equivalente a las tablas dinámicas en Excel. Como ejemplo, tomamos unos datos públicos correspondientes de la STPS, la serie estadística sobre la magnitud de la ocupación en el comercio, en comparación con la población ocupada total (trimestral).

Uso de las funciones apply, lapply, tapply y sapply III

```
datos <- read.csv(file = 'http://datosabiertos.stps.gob.mx/Datos/DIL/clave/Tasa_de_Ocupacion_en_el_Comercio.csv',
fileEncoding = "latin1", nrow = 4224)
head(datos)
```

```
  Periodo Trimestre Entidad_Federativa    Sexo Población_ocupada_en_el_comercio
1    2005         1           Nacional Hombres                           4155341
2    2005         1           Nacional Mujeres                           3986139
3    2005         1     Aguascalientes Hombres                             39813
4    2005         1     Aguascalientes Mujeres                             36001
5    2005         1    Baja California Hombres                            120173
6    2005         1    Baja California Mujeres                             84336
  Población_ocupada Tasa_neta_de_ocupación_en_el_comercio
1          26597801                           15.62287424
2          14843275                           26.85484841
3            246177                           16.17251002
4            145656                           24.71645521
5            768989                           15.62740169
6            398911                           21.14155789
```

```
#Vemos los tipos de datos
str(datos)
```

```
'data.frame': 4224 obs. of  7 variables:
 $ Periodo                              : int  2005 2005 2005 2005 2005 2005 2005 2005 2005 2005 ...
 $ Trimestre                            : int  1 1 1 1 1 1 1 1 1 1 ...
 $ Entidad_Federativa                   : chr  "Nacional" "Nacional" "Aguascalientes" "Aguascalientes" ...
 $ Sexo                                 : chr  "Hombres" "Mujeres" "Hombres" "Mujeres" ...
 $ Población_ocupada_en_el_comercio     : chr  "4155341" "3986139" "39813" "36001" ...
 $ Población_ocupada                    : chr  "26597801" "14843275" "246177" "145656" ...
 $ Tasa_neta_de_ocupación_en_el_comercio: chr  "15.62287424" "26.85484841" "16.17251002" "24.71645521" ...
```

```
class(datos$Población_ocupada_en_el_comercio) <- "numeric"
```

```
Warning in class(datos$Población_ocupada_en_el_comercio) <- "numeric": NAs introducidos por coerción
```

Algunas observaciones sobre calidad de datos

- El valor `NA` significa Not Available, se usa para representar datos faltantes o no disponibles. La función `is.na` se utiliza para verificar si un vector tiene datos faltantes

```
x <- c(1,2,3,NA,4,10)
is.na(x)
```

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

- El símbolo `NaN` significa *Not a Number*, `Inf` denota el infinito ∞ y `-Inf` es menos infinito, $-\infty$. Algunas funciones pueden devolver este resultado.

```
Inf + 0
```

```
[1] Inf
```

```
Inf*(-Inf)
```

```
[1] -Inf
```

- `NULL` es el objeto vacío o nulo. Su longitud siempre es 0.

```
    x <- NULL
    length(x)
```

```
[1] 0
```

Últimos detalles sobre coerción I

- Los modos se pueden ordenar por la cantidad de información que contienen:

$$\texttt{"NULL"} < \texttt{"logical"} < \texttt{"numeric"} < \texttt{"complex"} < \texttt{"character"} < \texttt{"list"}$$

- En general, los modos de los objetos se pueden cambiar sin perder información. Con operaciones aritméticas o de comparación entre objetos de diferentes modos se realiza, los objetos se convierten al mismo modo de tal forma que no se pierda información. A esto se le llama *coerción*.
- A las funciones que tienen como argumentos caracteres, se les pueden dar datos de *cualquier* modo.
- Para cada modo, hay tres funciones: para crear objetos de ese modo (`logical(10)`), para probar si objetos son de ese modo (`is.logical`), y para forzar objetos a tal modo (`as.logical`).

```
as.numeric("13")+ 12
[1] 25
as.character(13)
[1] "13"
is.logical("TRUE")
[1] FALSE
is.logical(2)
[1] FALSE
is.logical(as.logical(2))
[1] TRUE
```

Otras funciones importantes I

- La función `attach` y `detach` permiten acceder a los elementos de un data frame como series independientes, pero no permite modificaciones a las series.

```
datos[1,"Sexo"]
```

```
[1] "Hombres"
```

```
attach(datos) # deja entrar a los datos
Sexo[1:10]
```

```
 [1] "Hombres" "Mujeres" "Hombres" "Mujeres" "Hombres" "Mujeres" "Hombres"
 [8] "Mujeres" "Hombres" "Mujeres"
```

```
Sexo[1] <- "Mujeres"
Sexo[1:10] # cambia el valor de esta variable
```

```
 [1] "Mujeres" "Mujeres" "Hombres" "Mujeres" "Hombres" "Mujeres" "Hombres"
 [8] "Mujeres" "Hombres" "Mujeres"
```

```
datos[1,"Sexo"]  # pero no el dataframe
```

```
[1] "Hombres"
```

```
detach(datos) # ya no es accesible, y sólo se mantiene una copia de las variables
Sexo[1:10]    # que se cambiaron
```

```
 [1] "Mujeres" "Mujeres" "Hombres" "Mujeres" "Hombres" "Mujeres" "Hombres"
 [8] "Mujeres" "Hombres" "Mujeres"
```

Otras funciones importantes II

- `subset` toma un subconjunto de renglones de un dataframe.

```
A2010 <- subset(datos, datos$Periodo == "2010")
head(A2010)
```

```
     Periodo Trimestre Entidad_Federativa    Sexo
1321    2010         1           Nacional Hombres
1322    2010         1           Nacional Mujeres
1323    2010         1     Aguascalientes Hombres
1324    2010         1     Aguascalientes Mujeres
1325    2010         1    Baja California Hombres
1326    2010         1    Baja California Mujeres
     Población_ocupada_en_el_comercio Población_ocupada
1321                          4478623          28421331
1322                          4525262          17103008
1323                            46881            274223
1324                            40376            173151
1325                           138286            820298
1326                           115881            494120
     Tasa_neta_de_ocupación_en_el_comercio
1321                           15.75796362
1322                           26.45886618
1323                           17.09594017
1324                           23.31837529
1325                           16.85801989
1326                           23.45199547
```

- La función `merge` sirve para unir data frames en varias formas: una aplicación útil se utiliza para unir diferentes registros en bases de datos con un campo común.

Otras funciones importantes III

- La función `table` crea tablas de frecuencias (conteos) de variables categóricas

```
table(datos$Sexo)
```

```
Hombres Mujeres
   2112    2112
```

```
with(datos, table(Sexo, Periodo))
```

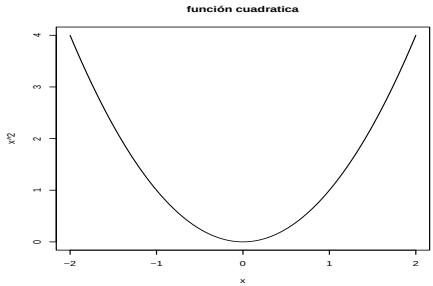
```
         Periodo
Sexo      2005 2006 2007 2008 2009 2010 2011 2012 2013 2014 2015 2016 2017 2018
  Hombres  132  132  132  132  132  132  132  132  132  132  132  132  132  132
  Mujeres  132  132  132  132  132  132  132  132  132  132  132  132  132  132
         Periodo
Sexo      2019 2020
  Hombres  132  132
  Mujeres  132  132
```

Curvas y funciones matemáticas

Usando R para hacer gráficas de funciones I

La función `curve` sirve para hacer gráficas de funciones matemáticas.

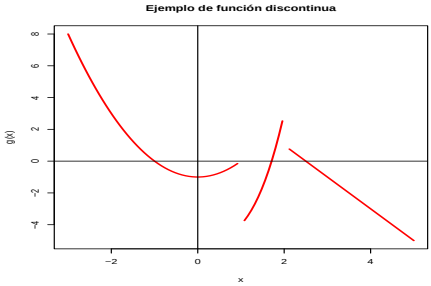
```
curve(x^2, from = -2, to = 2, main = "función cuadratica")
```



Podemos definir nuestras propias funciones, incluso con discontinuidades, como por ejemplo

Usando R para hacer gráficas de funciones II

```
g <- function(x){
                ifelse(( x < 1),x^2-1,ifelse((1<x & x<2),x^3-5, ifelse((x>2.1),5 - 2*x, NA)))
        }
curve(g(x), from = -3, to = 5, col = "red", lwd=3, main="Ejemplo de función discontinua")
abline(v = 0, h = 0)
```



Estructuras de Control

Control de ejecución: condicionales

Es en la creación de funciones de usuario, que se requiere un poco de conocimientos de programación. Estas se verán más adelante en ejemplos prácticos. Aquí se mencionan las principales estructuras.

- se puede usar: `if (cond)  else (cond)`
- Las condiciones pueden incluir operadores lógicos `&`, `|`.
- Otra posibilidad es la función `ifelse` sobre vectores.
- Para sustituir if's anidados, se usa la función `switch`.

Control de ejecución: ciclos

- Las funciones disponibles son `for, while, repeat`.
- `while (condicion.logica) instruccion`. Esta función termina cuando la condición es falsa.
- `for (variable.loop in valores) instruccion`
- `repeat (condicion) instruccion`.
- Para salir de los ciclos en cualquier punto, usamos `break`.
- Para saltar a la siguiente iteración, se usa `next`.

Gráficas

Introducción I

- Generar gráficas en R puede ser muy fácil o extremadamente complicado, ¡pero siempre es divertido!
- R produce el rango usual de gráficas estadísticas básicas, que incluye diagramas de dispersión, boxplots (gráficas de caja y brazo), histogramas, gráficas de pie, gráficas tridimensionales, animadas y muchas otras.
- El motor gráfico de R se concentra en algunos cuantos paquetes básicos (que están incluídos en la instalación original) y otros recomendados (que el usuario tiene que instalar, se marcan abajo con *):

  graphics contiene funciones de graficación para el sistema "base", incluye `plot, hist, boxplot` y muchas otras funciones. Una panorámica muy general de las gráficas puede verse con la instrucción `demo(graphics)`.

  * lattice contiene el código para generar gráficas Trellis (conocidas como gráficas condicionales), que son independientes del sistema base; incluye funciones como `xyplot, bwplot, levelplot`

  grid implementa un sistema de graficación independiente del sistema base. El paquete **lattice** está construido sobre **grid**. Rara vez las funciones de **grid** se llaman directamente.

Introducción II

grDevices contiene el código que implementa los diferentes dispositivos gráficos: X11, PDF, PostScript PNG, etc.

* ggplot2 Este es otro sistema basado en grid que interpreta y extiende las ideas de Leland Wilkinson de *The Grammar of Graphics*. Este es el paquete de moda, parte de `tidyverse`.

- La calidad de las gráficas de R permite que éstas estén listas para publicarse en artículos y libros.
- Se puede consultar la galería de gráficas de R para mayor detalle.

Decisiones iniciales

Cuando se hace una gráfica es importante tomar algunas decisiones iniciales, sobre todo porque cambios en la decisión pueden afectar la calidad de la imagen:

- ¿a qué dispositivo se enviará (impresora, pantalla)?
- ¿cuántos puntos habrá en la gráfica? (grandes cantidades, unos cuantos).
- ¿se requiere escalar la gráfica? Por ejemplo, si se usará en Word o PowerPoint.
- ¿Qué sistema de graficación se usará: base, grid/lattice, ggplot2? Usualmente no se pueden mezclar.
- Las gráficas del sistema Base se construyen con el principio del pintor: los objetos de la gráfica se agregan por capas.
- Las gráficas en el sistema ggplot2 también tienen capas o *layers*, pero son de conceptos.
- Las gráficas grid/lattice se crean de un sólo golpe: todos los parámetros se tienen que especificar de una sola vez en la función, y no se pueden agregar capas posteriores.

Gráficas del sistema base I

Este es el sistema más fácil y el primero que se creó, aunque ya no es el que se usa más comunmente: las gráficas se pueden programar y armar en etapas, agregando diferentes componentes como se necesiten.

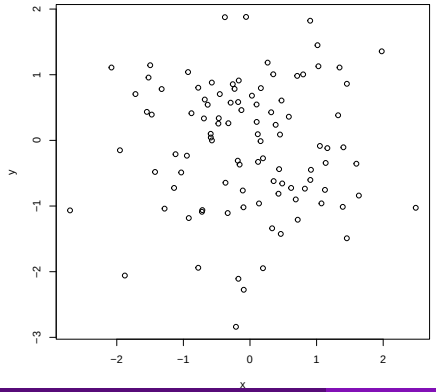
- El sistema base tiene muchos parámetros que se pueden modificar: están documentados en la función `par`
- Hay muchos acordeones para tener las funciones presentes.
- La función `par` se utiliza para especificar parámetros gráficos globales que afectan todas las gráficas en una sesión de R. Usualmente se pueden alterar como argumentos de algunas funciones específicas. Algunos ejemplos:
  - `pch`: **p**lotting **ch**aracter. Es el tipo de símbolo a usar en la gráfica.
  - `lty`: **l**ine **ty**pe (tipo de linea, continua, punteada, rayada, punto-raya, etc)
  - `lwd`: **l**ine **w**idth (ancho de línea, se especifica como un entero positivo)
  - `col`: **col**or, la función `colors()` da un vector de colores por nombre.
  - `las`: la orientación de las etiquetas en el eje x.
  - `bg`: el color del fondo de la gráfica.
  - `mar`: el tamaño del margen. Hay cuatro márgenes, en el órden: abajo, izquierda, arriba, derecha.
  - `oma`: el tamaño del margen exterior
  - `mfrow`: número de gráficas en un arreglo por renglón, columna (llenada por renglones)

Gráficas del sistema base II

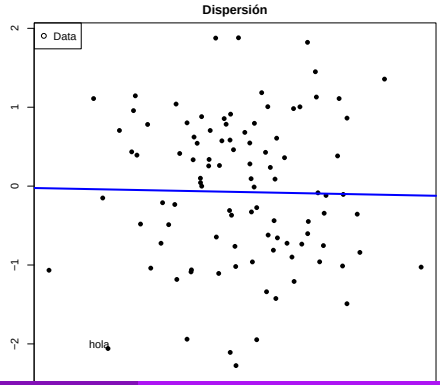
- `mfcol`: número de gráficas por renglón, columna (llenada por columna)

Algunos Ejemplos

```r
x <- rnorm(100); y <- rnorm(100)
plot(x,y) # función más básica
```

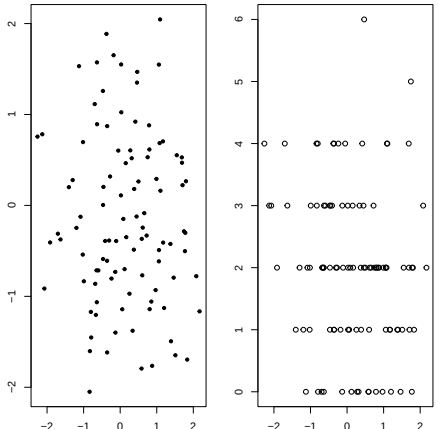



```r
par("mar") # el vector de márgenes, empezando por bottom, left, top, right.

[1] 5.1 4.1 4.1 2.1

par(mar=c(2,2,2,2))
plot(x, y, pch = 16); title("Dispersión"); text(-2, -2, "hola"); legend("topleft", le
fit <- lm(y ~ x) #ajusta una linea recta a los datos
abline(fit, lwd = 3, col = "blue")
```



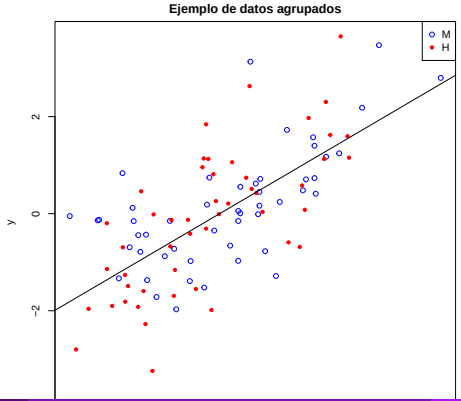
Ejemplos

```r
#genera dos gráficas en la misma hoja
x <- rnorm(100); y <- rnorm(100); z <- rpois(100,2) # genera datos artificiales
par(mar = c(2,2,2,2), oma = c(0,0,0,0))
par(mfcol = c(1,2)) # 1 renglón y dos columnas
plot(x, y, pch = 20)
plot(x, z, pch = 21)
```



Ejemplos

```
par(mfrow = c(1,1), mar = c(2,4,2,0.1))
x <- rnorm(100); y <- x + rnorm(100)
g <- gl(2, 50, labels = c("H", "M")) #genera un factor con dos niveles
plot(x, y, type = "n", main = "Ejemplo de datos agrupados")
points(x[g == "M"], y[g == "M"], col = "blue")
points(x[g == "H"], y[g == "H"],col="red",pch=20)
legend("topright", legend = c("M", "H"), pch = c(1, 20), col = c("blue", "red"))
fit2 <- lm(y ~ x)
abline(fit2)
```



Funciones básicas del sistema base

`plot` hace una gráfica de dispersión, u otros tipos de gráficas dependiendo del tipo de objeto que se grafica.

`lines` agrega lineas a una gráfica ya existente, dado un vector de valores `x` y el correspondiente vector de valores `y`. La función sólo conecta las lineas.

`points` agrega puntos a una gráfica ya existente.

`text` agrega etiquetas de texto a una gráfica usando coordenadas específicas `x,y`

`title` agrega anotaciones a las etiquetas de los ejes `x` y `y`, título, subtítulo, margen exterior

`mtext` agrega texto arbitrario a los márgenes (interior y exterior) de la gráfica

`axis` agrega ticks/etiquetas a los ejes.

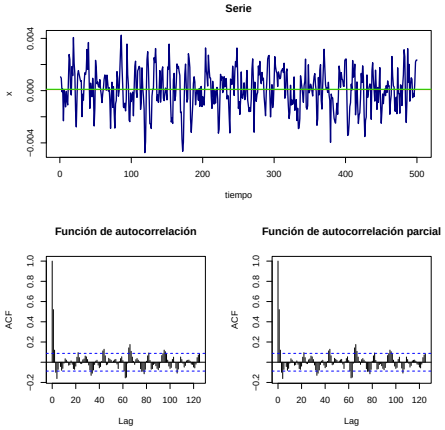
Ejemplos de una función que genera una gráfica

El siguiente código define una función que genera varias gráficas sobre una serie de tiempo

```
tsgraficas <- function(x, k = round(length(x)/4)){
                        oldpar <- par("oma")  # guarda los valores del parámetro oma
                        par("oma" = c(0.1,0.1,0.1,0.1))
                                              layout(matrix(c(1,1,2,3), nrow = 2, byrow = T)) #acomodo de las gráficas en el espacio
                                              plot.ts(x, lwd = 2, lty = 1, col = "navy",xlab = "tiempo", main="Serie")
                        abline(h = mean(x), col = "red")
                        abline(h = median(x), col = "green")
                        acf(x, main = "Función de autocorrelación", lag.max = k)
                        acf(x, main = "Función de autocorrelación parcial", lag.max = k)
                        par("oma" = oldpar)   # regresa los valores a su valor anterior
              }
```

Ejemplos de una función gráfica

```
x <- arima.sim(n=500,list(ar=c(0.8,-0.4),ma=c(-0.227,0.24)),sd=0.0013) #ejemplo de una serie de tiempo simulada
tsgraficas(x)
```



Aplicación. MoviLens: Un estudio de caso sobre películas

MovieLens I

- El conjunto de datos en el folder `ml-latest-small` describe calificaciones (ratings) de 5-estrellas y actividad de etiquetado de MovieLens, un servicio de recomendación de películas. Contiene 100,836 ratings y 3,683 etiquetas sobre las 9,742 películas. Estos datos fueron creados por 610 usuarios entre el 29/3/1996 y 24/9/2018. El conjunto de datos se generó el 26/9/2018.
- Los usuarios fueron seleccionados al azar por inclusión. Todos los usuarios seleccionados han calificado por lo menos 20 películas. No hay información demográfica. Cada usuario es representado por un identificador, y no se provee ninguna otra información.
  Los datos están contenidos en los arhivos `links.csv`, `movies.csv`, `ratings.csv` y `tags.csv`
- Haremos un análisis de estos datos, aprovechando para revisar cómo manipular datos, y como primer paso importaremos las bases de datos en formato `csv` a `R` para luego proceder a su análisis y ampliar el uso de algunas funciones de `R` para ese propósito.

Un estudio de caso

Ver el archivo markdown.