Conceptos básicos para programar en R

Prof: Gabriel Martos Venturini

UTDT

Este tutorial tiene como objetivo presentar las principales características de R; al final del mismo hay una hoja resumen de los principales comandos que aquí discutimos y que serán de utilidad a lo largo del curso. La experiencia indica que la única forma de aprender a programar (en cualquier lenguaje) es programando. En este sentido, te recomendamos realizar los ejercicios propuestos a lo largo de éste documento.

Si quieras revisar otro material con más explicaciones y ejemplos, puede recurrir al tutorial "R para principiantes" (disponible online en tanto en español como en inglés). Existen muchas referencias sobre R a nivel introductorio, intermedio y avanzado, si quieres o precisas más material, ponte en contacto con los docentes del curso para que te indiquen que recursos conviene consultar.

Descarga e instalación del programa

R es de uso GRATUITO y para descargarlo tenés que visitar la web de CRAN: http://cran.r-project.org. Te recomendamos instalar también RStudio: http://www.rstudio.com. Tené en cuenta que Rstudio es solo una plataforma sobre la que se ejecuta R (si no tienes instalado R en tu computadora, no podrás ejecutar ninguna instrucción).

Tipos de datos en R

Básicamente R funciona como una 'gran calculadora' donde podemos almacenar información e incluso crear nuestras propias 'funciones'. Cuando trabajemos en R a lo largo de esta materia, en general vamos a importar bases de datos externas, y con los datos en la memoria de R vamos a correr diferentes 'modelos' con los que resolvemos problemas de índole práctica. Antes de llegar a este punto, necesitamos entender que *tipos* de datos y que *estructuras* de datos soporta R; y aprender a **manipular** la información de cada una de estas estructuras dentro de R.

A grandes rasgos, en la memoria de R podemos almacenar 5 tipos de datos:

```
# 1) Lógicos (asumen los valores TRUE y FALSE):
l <- TRUE
print(class(l))
```

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

```
# 2) Numéricos (números reales):
n <- 23.5187
print(class(n))
```

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

```
# 3) Enteros (números naturales).
e <- 1L
print(class(e))
```

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

```
# 4) Complejos (números complejos)
d <- 3 - 2i
print(class(d))
```

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

```
# 5) Caracteres (cadenas de caracteres alfa-numéricos):
ca <- 'machine learning'
print(class(ca))
```

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

Los datos en la memoria de R se almacenan en diferentes *estructuras*. Dichas *estructuras* dependerán de la **dimensión** de los datos y de los tipos de datos con los que trabajes. La siguiente figura resume las características de éstas estructuras atendiendo a la 'dimensión' de los datos y la 'tipología' de los mismos:

	Homogeneous	Heterogeneous
1d	Atomic vector	List
2d	Matrix	Data Frame
nd	Array	

Figure 1: Tipos de datos en R

Por *Homogeneos* entendemos datos del mismo tipo (por ejemplo todos numéricos), y por *Heterogéneos* lo contrario (algunos datos son numéricos, otros caracteres, etc). En general en Estadística y ML (y también en Econometría, Economía, etc), algunas variables son numéricas y otras categóricas; por tanto en este curso utilizaremos con mayor frecuencia la estructura **data–frame**. Más adelante discutiremos como importar datos externos (fuera de R) a objetos como los descritos en la figura anterior. Ahora nos concentramos en discutir como crear y *manipular* datos en diferentes estructuras dentro de R.

Creando y manipulando vectores en R

El comando 'concatenar'

El comando **c()** en R nos permite 'vectorizar' (crear vectores) con datos del mismo tipo en R. Veamos algunos ejemplos (a lo largo de este curso puedes utilizar los símbolos "=" y "<-" de forma equivalente en R, aunque existen algunas diferencias entre estos dos 'operadores' que son importantes cuando quieres crear tus propias funciones en R):

```r
x <- c(1,2,3)
print(x)
```

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

```r
sem <- c('lunes','martes','miércoles','jueves','viernes')
print(sem)
```

```
## [1] "lunes"     "martes"    "miércoles" "jueves"    "viernes"
```

Si el/los vectore/s contienen datos numéricos, podemos hacer las operaciones habituales sobre estos elementos. R opera *punto a punto* sobre los vectores (y las matrices), salvo que le indiques lo contrario de manera explícita. Veamos algunos ejemplos:

```r
2*x + 1 # A cada elemento de x lo multiplico por 2 y le sumo 1.
```

```
## [1] 3 5 7
```

```r
x^2 # Cuadrado de todos los elementos de x.
```

```
## [1] 1 4 9
```

```r
sqrt(x) # Raíz de todos los elementos de x.
```

```
## [1] 1.000000 1.414214 1.732051
```

```r
log(x)  # Logaritmo de todos los elementos de x.
```

```
## [1] 0.0000000 0.6931472 1.0986123
```

```r
exp(x) # Exponencial de todos los elementos de x.
```

```
## [1]  2.718282  7.389056 20.085537
```

```r
x + sin(x) # A cada elemento de x le sumo el valor del seno de ese elemento....
```

```
## [1] 1.841471 2.909297 3.141120
```

Como puedes ver en los ejemplos anteriores, al igual que ocurre con una calculadora estándard, en R existen una gran cantidad de funciones pre-definidas (como el seno, el exponencial, etc). Con el comando concatenar '**c()**' también podemos unir (pegar) vectores con otros números u otros vectores (de dimensiones arbitrarias):

```r
c(9.2,8.5,7.1,x)
```

```
## [1] 9.2 8.5 7.1 1.0 2.0 3.0
```

```r
y <- 4:9
print(y)
```

```
## [1] 4 5 6 7 8 9
```

```r
z = c(x,y)
print(z)
```

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

Cuando las operaciones que hacemos son incoherentes o no definidas, el programa nos lanza un mensaje de error (en rojo) que nos advierte de la incoherencia de nuestra solicitud. Veamos que ocurre, por ejemplo, si intentamos multiplicar un vector con caracteres:

```r
2*sem
```

```
## Error in 2 * sem: argumento no-numérico para operador binario
```

```r
# Observa que al intentar concatenar vectores con diferentes tipos de datos no obtienes
# un mensaje de error; el elemento resultante es transformado en tipo caracter:
c(x,sem)
```

```
## [1] "1"         "2"         "3"         "lunes"     "martes"    "miércoles"
## [7] "jueves"    "viernes"
```

```r
# R transforma los elementos numéricos en texto para que la operación se pueda realizar.
```

ES MUY IMPORTATE que aprendas a leer los mensajes de error en la consola, y a buscar ayuda googleando para comprender el motivo de tu error. Existen algunos comandos que resultan de bastante utilidad en la práctica. Por ejemplo el comando **lenght()** que me dice cuantos *elementos* (que dimensión) tiene un vector.

```r
length(x) # x tiene 3 elementos, luego:
```

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

```r
z = 1:10
length(z) # z tiene 10 elementos, luego:
```

```
## [1] 10
```

Ejercicios en clase:

1- Con el comando `sum()` puedes sumar los elementos de un vector. Crea un vector con los elementos $1^2, 2^2, \ldots, 10^2$ y calcula la media y la varianza de los datos en el vector (contrasta tus resultados con los que obtienes utilizando las funciones `mean()` y `var()`.

2- Crea otro vector con los elementos $2^1, 2^2, \ldots, 2^{10}$, y determina comparando contra los elementos del vector del inciso anterior para que valores de $n \in \{1, 2, \ldots, 10\}$ se cumple que: a) $2^n < n^2$, b) $2^n = n^2$ y c) $2^n > n^2$ (compara los vectores con los operadores "`==`", "`<`" y "`>`").

Acceso y manipulación de los elementos de un vector:

En varios contextos (cuando hacemos uso de un loop for) necesitamos acceder a *elementos* que ocupan una posición específica dentro de un vector. Para ello usamos el comando []:

```r
z = c('a','e','i','o','u')
print(z)
```

```
## [1] "a" "e" "i" "o" "u"
```

```r
z[1]  # Extraemos el primer elemento de z.
```

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

```r
z[-1] # Extraemos todos los elementos menos el primer elemento de z.
```

```
## [1] "e" "i" "o" "u"
```

```r
z[c(1,3,5)] # Extraemos los elementos que ocupan posiciones impares en z.
```

```
## [1] "a" "i" "u"
```

```r
z[-c(1,5)] # Extraemos todos los elementos menos el primero y el último.
```

```
## [1] "e" "i" "o"
```

A lo largo del curso, en repetidas ocasiones vamos a necesitar *buscar elementos de un vector que cumplen con alguna condición* pre-especificada (ej: cuando analicemos los resultados de la validación cruzada y busquemos la configuración de los hiper–parámetros de un modelo que minimizan el error sobre una muestra de test).

Para ello utilizamos la función **which()** más una condición lógica que indique lo que estamos buscamos. Por ejemplo, si quisiéramos saber que posición ocupa la letra 'o' en un vector de vocales:

```
which(z=='o') # de los elementos de z, que posición ocupa aquel que es '==' (igual a) la letra 'o'
```

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

```
x = 1:10
which(x<=3) # de los elementos de x, que posición ocupan aquellos que son menores o iguales a 3
```

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

Podemos combinar condiciones de búsqueda usando conectores lógicos: "o" que en R se escribe con el símbolo "|" e "y" que en R se escribe con el símbolo "&". Veamos algunos ejemplos:

```
print(z)
```

```
## [1] "a" "e" "i" "o" "u"
```

```
which( z == 'o' | z == 'a' ) # De los elementos de z, cual es '==' (igual a) la letra 'o' o 'a'.
```

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

```
x = -5:5
print(x)
```

```
##  [1] -5 -4 -3 -2 -1  0  1  2  3  4  5
```

```
which(x>1 & x<=3) # De los elementos de x, cuales son mayores a 1 y menores o iguales que 3.
```

```
## [1] 8 9
```

También nos puede interesar **modificar** algún/algunos valores particulares de un vector. Por ejemplo, quiero cambiar la vocal 'u' (que ocupa la posición 5 dentro del vector **z**) por la letra 'm':

```
print(z)
```

```
## [1] "a" "e" "i" "o" "u"
```

```
z[5] = 'm' # cambiamos 'u' por 'm'.
print(z)
```

```
## [1] "a" "e" "i" "o" "m"
```

```
print(x)
```

```
##  [1] -5 -4 -3 -2 -1  0  1  2  3  4  5
```

```
x[which(x>2)] = 2 # De los elementos de x, aquellos que fueran mayor que 2 son truncados en 2.
print(x)
```

```
##  [1] -5 -4 -3 -2 -1  0  1  2  2  2  2
```

Funciones aplicadas sobre vectores:

Las funciones son comandos de R que o bien ya vienen pre-programados o que creamos para repetir una tarea llamando a la función cuando resulte conveniente. La estructura general de las funciones (ya veremos más adelante como crear nuestras propias funciones) es muy intuitiva: *fun('objeto sobre el que se aplica', 'parámetro/s de la función')*. Por ejemplo, en algunos contextos resulta interesante 'redondear' números (hacia abajo o hacia arriba), para ello utilizamos la función *floor*, *ceiling*, *round* y *trunc*:

```
x = c(-1.90,-1.46,2.36,2.92)
print(x)
```

```
## [1] -1.90 -1.46  2.36  2.92
```

```
floor(x) # Redondeo 'hacia abajo'
```

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

```
ceiling(x) # Redondeo 'hacia arriba'
```

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

```
round(x,1) # Redondeo 'a 1 decimal' (este es un parámetro de la función round que puedo cambiar!)
```

```
## [1] -1.9 -1.5  2.4  2.9
```

```
trunc(x) # Trunca el número quitando la parte decimal
```

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

También nos puede interesar calcular *sumas*, *sumas acumuladas*, *valores absolutos*, *maximos*, *mínimos* u *ordenar* los elementos de un vector:

```
x = c(-1.90,-1.46,2.36,2.92)
print(x)
```

```
## [1] -1.90 -1.46  2.36  2.92
```

```
sum(x) # Suma de los elementos de x
```

```
## [1] 1.92
```

```
cumsum(x) # Suma acumulativa de los elementos de x
```

```
## [1] -1.90 -3.36 -1.00  1.92
```

```
abs(x) # Devuelve los valores absolutos de los elementos de x
```

```
## [1] 1.90 1.46 2.36 2.92
```

```
max(x) # Máximo
```

```
## [1] 2.92
```

```
min(x) # Mínimo
```

```
## [1] -1.9
```

```
sort(x,decreasing = T) # Ordenamos 'descendentemente' (parámetro de la función!) los elementos de x.
```

```
## [1]  2.92  2.36 -1.46 -1.90
```

En el apéndice tienes 3 listados con los comandos y funciones básicas para minpular datos, hacer estadística descriptiva y utilizar algunas herramientas de programación básica (que discutirmos con algo de detalles en la próxima clase).

Para cerrar este apartado de vectores, vamos a describir como hacer operaciones algebraicas con vectores (las mismas que hacías en mate I). Para esto recordemos el concepto de producto interior y norma: Sean $\mathbf{v} = (v_1, \ldots, v_n)$ y $\mathbf{w} = (w_1, \ldots, w_n)$ dos vectores en $\mathbb{R}^n$, el producto interior se define como: $\mathbf{v}^T\mathbf{w} = \sum_{i=1}^{n} v_i w_i$, y la norma $\|\mathbf{v}\| = \sqrt{\mathbf{v}^T\mathbf{v}}$. En R podemos computar estas magnitudes de la siguiente manera:

```
v = c(0,5,-1)
w = c(1/2,-2,7)

# Producto interior:
t(v)%*%w  # v'*w (notar que esto es diferente de v*w)
```

```
##      [,1]
## [1,]  -17
```

```
# Normas:
(t(v)%*%v)^0.5
```

```
##         [,1]
## [1,] 5.09902
```

```
(t(w)%*%w)^0.5
```

```
##         [,1]
## [1,] 7.29726
```

```
# Ejercicio: Calcula el ángulo entre los dos vectores del ejemplo.
```

Creando y manipulando matrices en R

Una matriz es un arreglo bidimensional de números, en general las denotamos con letras negritas mayúsculas:

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix}$$

Las matrices pueden entenderse en R como objetos bi-dimensionales y por lo tanto habrá dos indices para indicar la posición de cada elemento en este objeto: Uno para indexar a que *fila* y el otro para indexar a que *columna* pertenece el elemento en cuestión. Es importante mencionar que los elementos que almacenamos dentro de una matriz tienen que ser todos del mismo tipo (esto quiere decir que no podemos tener números y caracteres al mismo tiempo dentro de una matriz).

El comando 'matrix'

Con el comando **matrix(*datos*, *nrow*, *ncol*, byrow = F)** creamos matrices en R

```
x <- c(1,2,3,4)
matrix(x, ncol = 2, nrow =2, byrow = F) # byrow = F -> reparte los datos en x por columnas.
```

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

```
matrix(x, ncol = 2, nrow =2, byrow = T) # byrow = T -> reparte los datos en x por filas.
```

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

Operaciones algebraicas y funciones sobre matrices

Al igual que con los vectores, podemos hacer operaciones punto a punto y aplicar funciones (ej: cuadrado, seno, logaritmo, etc) sobre los elementos que constituyen una matriz. Veamos algunos ejemplos de como opera R con estos elementos:

```
A = matrix(c(1:9),ncol=3)
print(A)
```

```
##      [,1] [,2] [,3]
## [1,]    1    4    7
## [2,]    2    5    8
## [3,]    3    6    9
```

```
A + 2*A # Suma de dos matrices
```

```
##      [,1] [,2] [,3]
## [1,]    3   12   21
## [2,]    6   15   24
## [3,]    9   18   27
```

```
A*A # Producto elemento a elemento de A (cuadrado de los elementos de A)
```

```
##      [,1] [,2] [,3]
## [1,]    1   16   49
## [2,]    4   25   64
## [3,]    9   36   81
```

```
A^2 # Cuadrado de los elementos de A
```

```
##      [,1] [,2] [,3]
## [1,]    1   16   49
## [2,]    4   25   64
## [3,]    9   36   81
```

```
sqrt(A) # Raíz de los elementos de  A
```

```
##          [,1]     [,2]     [,3]
## [1,] 1.000000 2.000000 2.645751
## [2,] 1.414214 2.236068 2.828427
## [3,] 1.732051 2.449490 3.000000
```

```
log(A)  # Logaritmo de los elementos de  A
```

```
##           [,1]     [,2]     [,3]
## [1,] 0.0000000 1.386294 1.945910
## [2,] 0.6931472 1.609438 2.079442
## [3,] 1.0986123 1.791759 2.197225
```

```
exp(A) # Exponencial de los elementos de  A
```

```
##           [,1]      [,2]     [,3]
## [1,]  2.718282  54.59815 1096.633
## [2,]  7.389056 148.41316 2980.958
## [3,] 20.085537 403.42879 8103.084
```

```
sin(A) # Seno de los elementos de  A
```

```
##           [,1]       [,2]      [,3]
## [1,] 0.8414710 -0.7568025 0.6569866
## [2,] 0.9092974 -0.9589243 0.9893582
## [3,] 0.1411200 -0.2794155 0.4121185
```

Las funciones de *suma*, *valores absolutos*, *maximos*, *mínimos* también se pueden emplear con una matriz:

```
sum(A) # Devuelve la suma de todos los elementos de A
```

```
## [1] 45
```

```
max(A) # Devuelve el máximo entre todos los elementos de A
```

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

```
min(A) # Devuelve el mínimo entre todos los elementos de A
```

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

En algunos contextos puede resultar de utilidad aplicar alguna de estas funciones por filas o por columnas sobre los datos de una matriz. En ese caso utilizamos el comando *apply(matriz, 'margen', 'fun')*, como ves en el siguiente ejemplo:

```
print(A)
```

```
##      [,1] [,2] [,3]
## [1,]    1    4    7
## [2,]    2    5    8
## [3,]    3    6    9
```

```
apply(A,1,sum) # Devuelve la suma por columnas de A.
```

```
## [1] 12 15 18
```

```
apply(A,2,min) # Devuelve los mínimos por filas de A.
```

```
## [1] 1 4 7
```

Reorganizando los datos en una o más matrices

Para *concatenar matrices* (apilar por filas o columnas un conjunto de matrices), utilizamos los comandos `cbind` y `rbind`, veamos un ejemplo:

```
print(A)
```

```
##      [,1] [,2] [,3]
## [1,]    1    4    7
## [2,]    2    5    8
## [3,]    3    6    9
```

```
B = -1*A
print(B)
```

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

```
cbind(A,B) # cbind = column-bind (unir por columnas)
```

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

```
rbind(A,B) # rbind = row-bind (unir por filas)
```

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

Con el comando **dim()** visualizamos las *dimensiones* (el m y el n) de una matriz:

```
C=matrix(c(1:8),ncol=2)
print(C)
```

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

```
dim(C) # El primero número indica cantidad de filas y el segundo de columnas.
```

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

En algunos contextos necesitamos acceder (o modificar) ciertos *elementos* de una matriz. Como lo hacemos? Al igual que con los vectores usamos el operador [], la única diferencia es que ahora necesitamos especificar tanto las filas como las columnas a las que queremos acceder. Veamos un primer ejemplo:

```
print(C)
```

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

```
C[2,1] # Nos devuelve el elemento que ocupa la posición de la segunda fila y la primera columna.
```

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

Al igual que con los vectores, podemos utilizar de varias maneras el operador:

```
C[-c(1,4), ] # Quitamos las filas 1 y 4 y nos quedamos con todas las columnas.
```

```
##      [,1] [,2]
## [1,]    2    6
## [2,]    3    7
```

Ejercicios:

1- Construye en R la siguientes matrices:

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 6 & 5 & 4 \\ 7 & 8 & 9 \end{bmatrix} \qquad B = \begin{bmatrix} 6 & 5 & 4 \\ 1 & 2 & 3 \\ 9 & 8 & 7 \end{bmatrix}$$

extraer la primera fila de A y la segunda columna de B. Luego verifica (y explica) que resultado obtienes al aplicar las siguientes funciones:

- c(A)
- A <= 5
- A > B
- t(B)
- diag(A)
- rowSums(B)
- colSums(A)
- apply(A, 1, max)
- apply(A, 2, max)
- apply(B, 1, mean)

- apply(B, 2, sd) (sd = comando para calcular la desviación standard de un conjunto de datos)

Data Frames

Los **data frames** son la estructura de datos que más utilizamos en la práctica. A diferencia de las matrices, podemos almacenar en ellos elementos de dirente naturaleza (algunas columnas con números y otras con caracteres). Al igual que las filas y columnas de una matriz, todos los elementos de un data frame deben ser de la misma longitud. De este modo, pueden usarse funciones tales como *dim*, *apply*, *rbind*, etc, sobre un data frame como si se tratara de una matriz. Operamos con y accedemos a los elementos de un data frame de la misma manera que lo hacemos con las matrices. Creemos un data frame de juguete (simulando datos de pesos, alturas y club de fútbol de algunos estudiantes del curso):

```
peso   <- c(60.1,75.5,82.9,78.5,69.6,50.4)
altura <- c(151,174,181,175,164,141)
club   <- c('River','Boca','Racing','Talleres','River','Independiente')

Datos = data.frame(peso, altura, club)

print(Datos)
```

```
##   peso altura          club
## 1 60.1    151         River
## 2 75.5    174          Boca
## 3 82.9    181        Racing
## 4 78.5    175      Talleres
## 5 69.6    164         River
## 6 50.4    141 Independiente
```

```
str(Datos) # En el mismo objeto 'Datos' hay diferentes "tipos" de datos.
```

```
## 'data.frame':    6 obs. of  3 variables:
##  $ peso  : num  60.1 75.5 82.9 78.5 69.6 50.4
##  $ altura: num  151 174 181 175 164 141
##  $ club  : chr  "River" "Boca" "Racing" "Talleres" ...
```

Para acceder a los diferentes elementos dentro de un data frame podemos utilizar el mismo comando que en el caso de las matrices, sin embargo y desde el punto de vista práctico cuando trabajamos con datos almacenados en *data-frames*, será mas cómodo hacer referencia a las columnas *por su nombre* (en particular cuando por ejemplo leas datos externos y quieras hacer regresiones con las variables de esos datos). Para ello necesitamos indicarle a R que queremos trabajar de esta forma; y por ello utilizamos el **attach()** sobre el data frame para luego extraer las variables con el signo '$'. Veamos un ejemplo:

```
# Queremos extraer la columna de altura del data--frame:
Datos[,2]

# Alternativamente:
attach(Datos)
Datos$altura

## Ejercicios:
# 1- Utiliza los comandos: summary(Datos) y table(Datos$club).
# 2- Investiga para sirve el comando **detach()** en R.
```

En algunos contextos nos interesa trabajar solo con subconjuntos de la información que disponemos dentro de un data-frame. Por ejemplo, para el data frame que acabamos de crear, nos puede interesar analizar solamente a los hinchas de River. El comando *subset()* nos ayuda a realizar este tipo de operaciones de

manera simple en R:

```
Datos.River = subset(Datos, club=='River', select=c(altura,peso) )
print(Datos.River)
```

```
##   altura peso
## 1    151 60.1
## 5    164 69.6
```

Importando datos externos en formato dataframe:

En general utilizamos R para analizar datos (y obviamente no los vamos a cargar a mano). Leer datos externos (en virtualmente cualquier formato) es muy simple. La función habitual para ello es el comando `read.table()`:

```
datos = read.table('C:\...\tusdatos.txt', sep = ',', dec = '.', header = T)
```

Por defecto, los datos en formato diferentes a *.RData* se leen y almacenan internamente en un data frame (luego puedes reconvertir parte o el total de los datos a otro formato si así lo deseas). Con el comando \texttt{read.table() puedes leer muchos formatos de datos distintos (.txt, .csv, etc). Si quieres levantar datos en formatos poco extendidos (ficheros de Stata, EViews, SPSS, etc), R tiene un montón de librerías que te permiten leer estos datos utilizando una simple línea de comando.

Arrays y Listas

Los arrays son *matrices 3-dimensionales* (imagina una matriz en 3 dimensiones). Se utilizan bastante poco en la práctica. Las listas constituyen el concepto más general de almacenamiento de datos, aquí adentro podemos coleccionar: variables, vectores, matrices y/o data frames (cada objeto de la lista puede ser cualquiera de los anteriores elementos). Veamos un ejemplo:

```
MiLista = list(a = 1, x = c(1,2,3), A = matrix(1:4,ncol=2), Datos)

MiLista[1]
```

```
## $a
## [1] 1
```

```
MiLista[2]
```

```
## $x
## [1] 1 2 3
```

```
MiLista[3]
```

```
## $A
##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4
```

```
MiLista[4]
```

```
## [[1]]
##   peso altura          club
## 1 60.1    151         River
## 2 75.5    174          Boca
## 3 82.9    181        Racing
## 4 78.5    175      Talleres
## 5 69.6    164         River
```

```
## 6 50.4    141 Independiente
```

A veces necesitamos acceder a algunos subconjuntos dentro de alguno de los objetos de la lista. Por ejemplo, supongamos que queremos la segunda columna de la matriz A (que ocupa el tercer lugar en *MiLista*), luego usamos dos veces el comando *[ ]* como sigue:

```
MiLista[[3]][,2]
```

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

Las listas suelen resultar útiles en algunos contextos de programación, pero en general las utilizamos solo para **guardar** resultados numéricos de distinto tipo en un solo objeto.

Apéndice

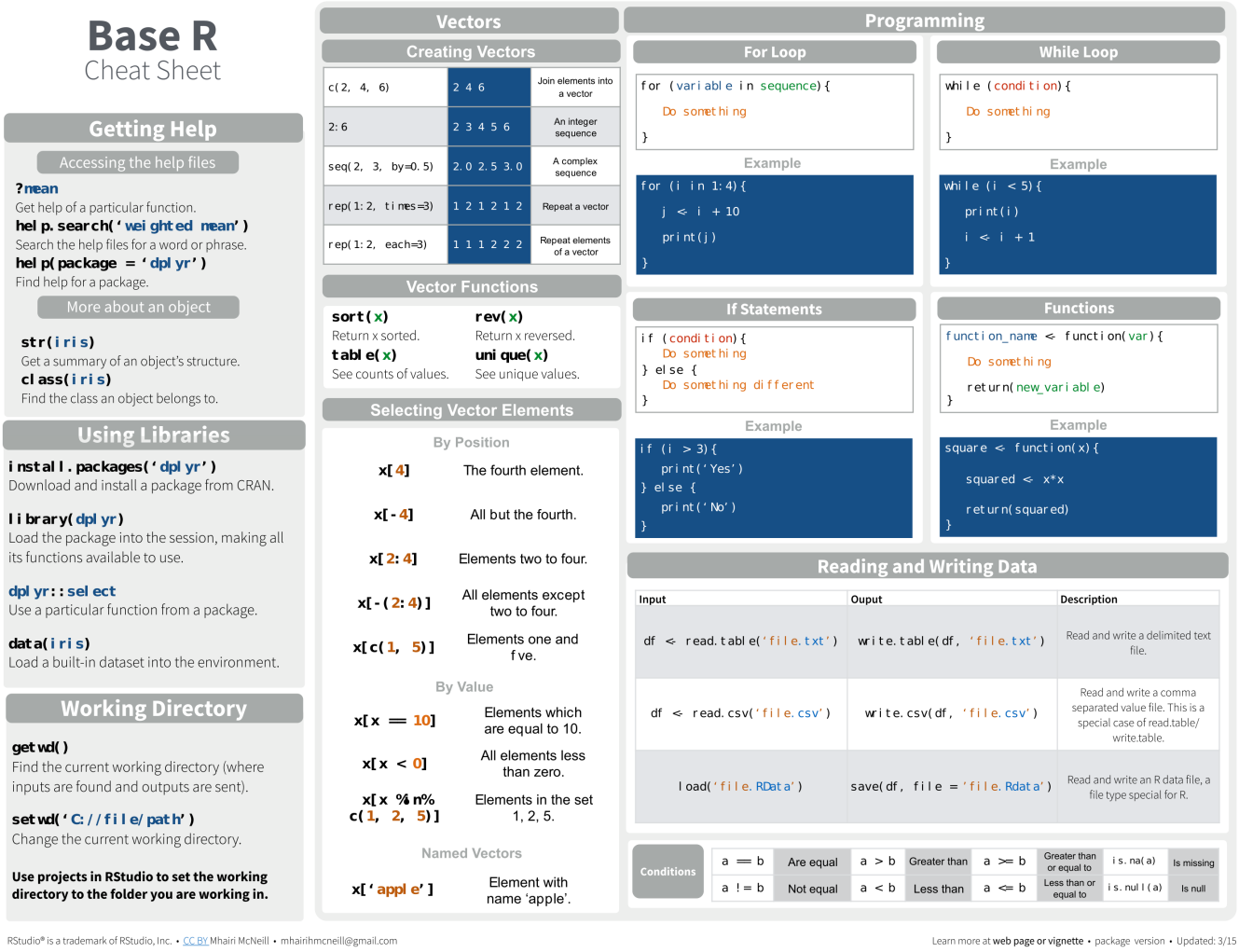

Figure 2: Funciones básicas para manipular datos y elementos dentro de R

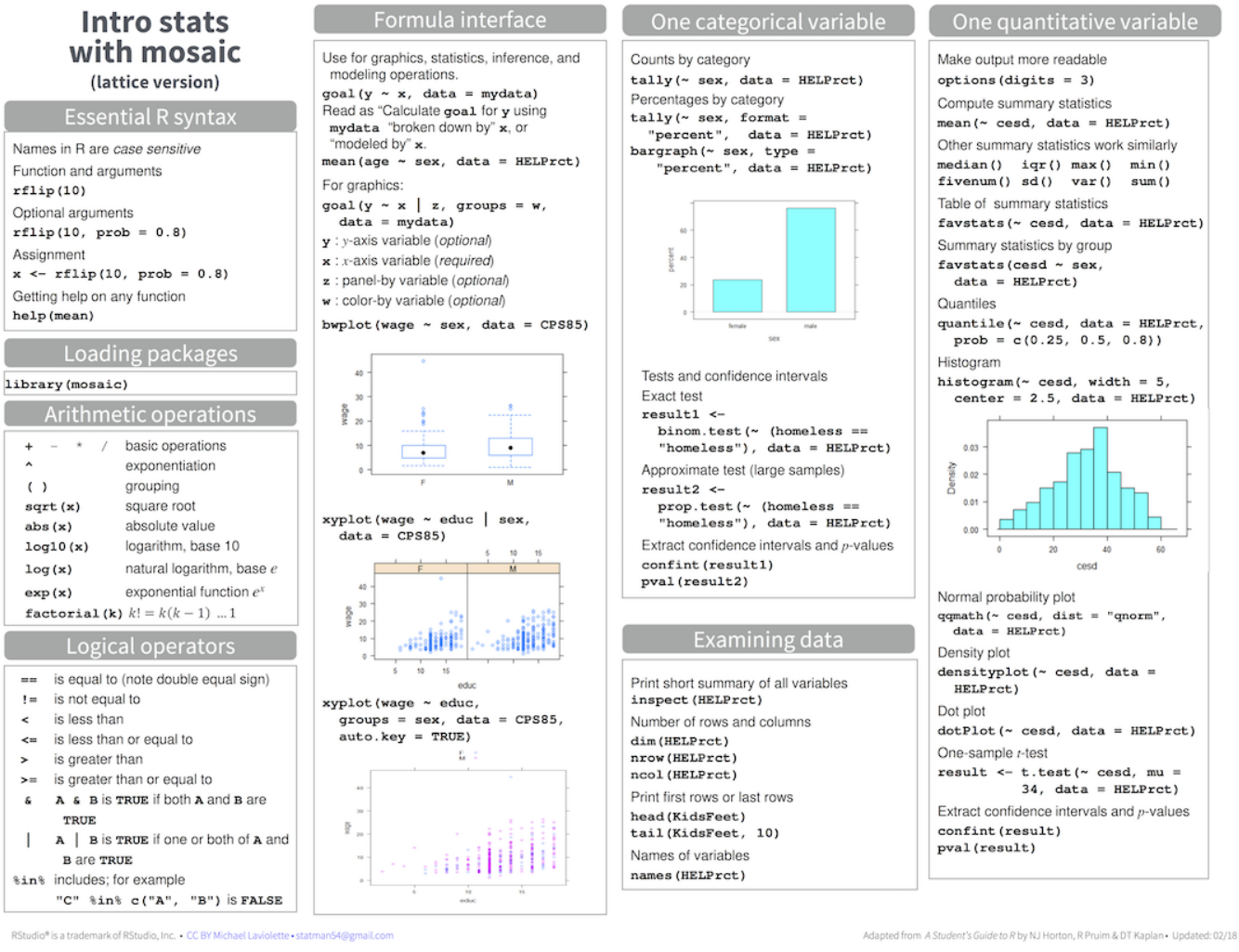

Figure 3: Funciones básicas para hacer análisis estadístico en R

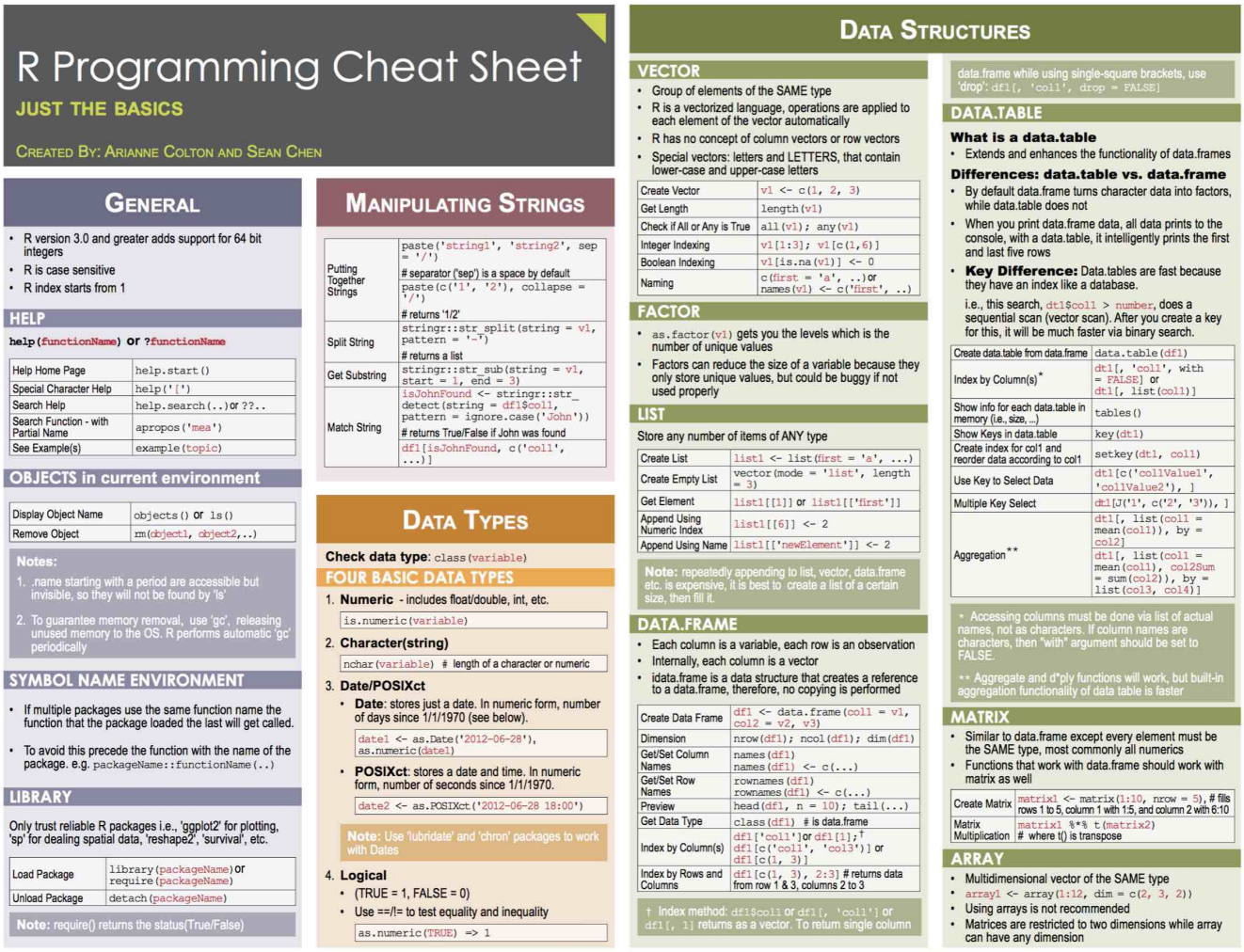

Figure 4: Funciones básicas para programar en R