CAPÍTULO 1 ESTRUCTURAS ATÓMICAS 3

1.1. Vectores 3
1.1.1. Acceder y extraer los elementos 5
1.1.2. Operaciones matemáticas 7
1.1.3. Reciclado 13

1.2. Atributos 14
1.2.1. Ejemplos 14

1.3. Matrices 15
1.3.1. matrix() 15
1.3.2. cbind(), rbind() 15
1.3.3. Acceder y extraer elementos 16
1.3.4. Operaciones matemáticas 16

1.4. Arrays 18
1.4.2. Operaciones con arrays 20

CAPÍTULO 2 22

2.1. Listas 22
2.1.1. Acceder y extraer elementos 23
2.1.2. Operaciones con listas 24

2.2. Dataframes y tibbles 26
2.2.1. Crear dataframes y tibbles 26
2.2.2. Acceder y extraer elementos 28

CAPÍTULO 3 30

3.1. Factores 30
3.1.1. Ejemplo con tipos ordinales 30
3.1.2. Ejemplo con tipos categóricos 30

3.2. Funciones 32
3.2.1. Argumentos por defecto 32
3.2.2. Crear funciones 34

Parte II Estructuras de datos

R opera con estructuras de datos que sirven para agrupar, almacenar y procesar múltiples elementos. Las principales estructuras de datos en R son: vector, matriz, array, lista, marco de datos, factor y función. Una estructura de datos es un objeto en R. Los objetos tienen atributos, por ejemplo: names, dimensions, dimnames y classes. El uso de estos atributos se irá viendo a medida que avance el curso.

Conviene ponerse de acuerdo sobre qué se llama dato. Hay dos tipos de datos: cuantitativo y categórico. Un dato cuantitativo responde a preguntas como ¿Cuánto hay? y resulta de una medida o un conteo, que se representa con números enteros, o sea una secuencia discreta, o un número real, es decir una secuencia continua. Un valor cuantitativo puede corresponder a dos escalas numéricas, de intervalo y racional. La escala de temperatura es cuantitativa de intervalo, pues define un orden y también la separación entre unidades, pero carece de un nivel de referencia 0. En cambio, la escala cuantitativa racional incluye un nivel de referencia 0, lo cual permite establecer cocientes. Un dato cuantitativo se puede manipular con técnicas matemáticas.

Por su parte, un dato categórico, también conocido como cualitativo, representa una categoría, un grupo claramente distinguible de otros grupos. Un dato categórico puede corresponder a las escalas ordinal y nominal. La escala nominal se representa con etiquetas, o nombres; varón-mujer, alto-bajo. No representan un ordenamiento y tampoco indican una diferencia específica en valor. La escala ordinal tambien se representa con etiquetas o nombres, pero incluye un ordenamiento; por ejemplo, baja velocidad, velocidad moderada, velocidad elevada, o un ranking de aceptación: 1 2 3 4 5. Pero la separación entre 1 y 2 y entre 3 y 4 no son necesariamente iguales. R clasifica las variables categóricas como factores y brinda técnicas especiales para manipularlas.

R opera con seis tipos de datos: character, numeric, integer, logical y complex. character incluye una letra aislada o un conjunto de letras, que en este curso se denomina cadena. El tipo numeric abarca los números reales; integer los números enteros, que en R se distinguen por el sufijo L (2L difiere de 2; aunque no se coloque un punto R interpreta el 2 como real); logical resuelve en TRUE o FALSE; complex refiere a números complejos. Más adelante se ve cómo reconocer los tipos de datos.

Capítulo 1 Estructuras atómicas

Ver videos Estructuras_1.mp4 y Estructuras_2.mp4.

En muchos casos es necesario trabajar con más de un tipo de dato. Quizá lo más frecuente es que se manipulen datos de tipo numérico, apto para cálculos matemáticos, pero muchos conjuntos de datos incluyen caracteres y cadenas de caracteres, por ejemplo, y es conveniente tener a disposición modos de operar con ellos. R ofrece esta prestación pero con condiciones. Hay estructuras que solamente aceptan albergar datos de un mismo tipo. Estas estructuras se denominan atómicas en referencia a que los tipos de datos que contiene no son subdivisibles, y son las que se ven en este capítulo. Vectores, matrices y arrays pertenecen a esta categoría.

1.1. Vectores

Un vector se puede visualizar como una serie de celdas contiguas conteniendo datos. Hay cinco tipos básicos de vectores: lógico, entero, real, complejo y cadena, designados en base al tipo de elemento que lo conforma. Un vector debe necesariamente estar compuesto por un único tipo de elemento. Así, un vector puede incluir sólo letras, o sólo números reales, pero no letras y números reales. En R este tipo de estructura se denomina atómica. Los vectores son la piedra fundamental de las estructuras de datos en R. El diagrama esquematiza la organización de los vectores.

Un vector se caracteriza por el tipo de dato que contiene, por el número de elementos que lo componen y por los atributos que lo caracterizan. R ofrece cuatro opciones para crear vectores: el operador ':' y las funciones 'c()', 'seq()'

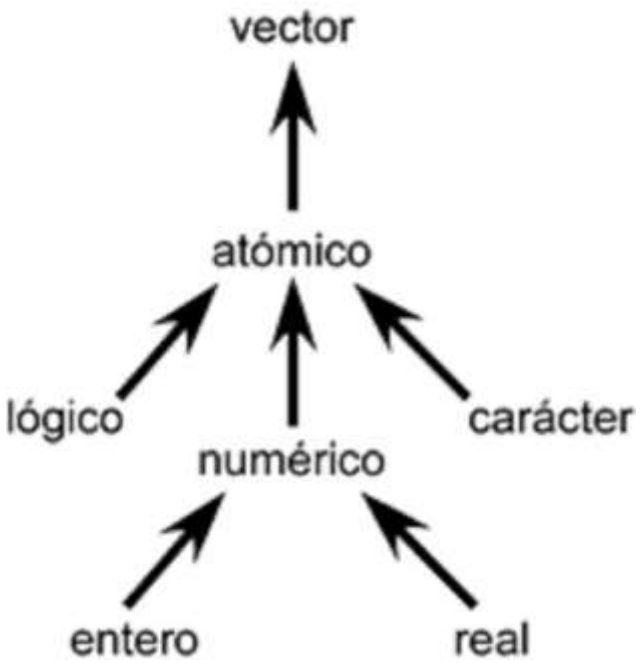

Figura 1.1: *Estructura de un vector atómico.*

y ’rep()’. El operador ’:’ se emplea así:

> 1:10

> -5:5

La función c() concatena los elementos que han de conformar el vector:
c(elemento1, elemento2, ..., elementoN, vector_x)
Ejemplos

> c(3, 4, 5)

> c(’este’, ’es’, ’un’, ’vector’)

> c(1:5, 6, 7)

La función seq (secuencia) opera solamente con elementos numéricos y tiene la siguiente estructura:
seq(from = valor inicial, to = valor final, by = salto). La omisión de by = presupone 1.
Ejemplos

> seq(3, 9)

> seq(9, 3) invierte el orden

> seq(from = 3, to = 9, by = 2) saltea uno.

Una variante permite predefinir el número de elementos. Comparar:

> seq(9, 3)

que devuelve 7 elementos con

> seq(9, 3, length.out = 10)

que devuelve 10, debiendo para ello recurrir a fracciones decimales.

Por último está la función rep(repetición):

rep(elemento a repetir, veces a repetir, ...)

que acepta elementos numéricos o de caracteres, y también vectores.

Ejemplos

> rep('uno', 5)

> rep(4, 6)

> rep(1:4, 5) se repite un vector

> rep(4:6, each = 3) repite cada elemento del vector.

En los ejemplos recién vistos la ejecución del código transfiere el resultado a la pantalla sin guardarlo en memoria. Si se desea emplear el resultado en otra operación es necesario asignarlo a una variable (la cual se mostrará en Global Environment). Por ejemplo, la ejecución de: rep_vect <- rep(1:4, 5) no arroja resultado en pantalla, pues el resultado ha sido almacenado en la variable rep_vect. Escribir:

> rep_vect para ver el resultado.

Teniendo el resultado como variable se puede operar con él. Escribir

> rep_vect * 2 para duplicar el vector.

Se puede explicitar qué representan los elementos de un vector asignándoles nombres. Crear el vector

> precio_fruta <- c(banana = 50, naranja = 40, pera =80)

Cada dato tiene un identificador. Explorar con:

> attributes(precio_fruta) devuelve que el vector tiene un atributo names. y con

> names(precio_fruta) se recuperan los nombres dados.

Se vieron los modos de crear vectores y de operar con ellos empleando todos sus elementos. Pero ¿cómo operar con sólo algunos de los elementos? Para ello es necesario poder acceder a los elementos individuales.

1.1.1. Acceder y extraer los elementos

Si se desea trabajar con ciertos elementos de un vector y no otros es necesario acceder a ellos individualmente. En R un vector no es sólo un conjunto de elementos, es un conjunto ordenado de elementos. Que R mantiene un registro de la posición de cada elemento se manifiesta en que precede la salida con [1]. Cada elemento de un vector tiene asociado un índice mediante el cual se lo identifica, y dicho índice se escribe entre []. Escribir

> a <- c(5, 6, 9, 44) y luego

> a[2]

para seleccionar un elemento. Y para ejecutar una operación matemática hacer

> a[2] * a[3].

Se remarca que en R no es válido un índice 0; se parte de 1.

R incluye algunas constantes ya definidas con estructura de vector.

> letters contiene las 26 letras del alfabeto en minúsculas y

> LETTERS las devuelve en mayúsculas.

> LETTERS[5] devuelve la quinta letra, E.

> LETTERS[1:5] devuelve las cinco primeras letras.

Otro modo de acceder a elementos de un vector es llamándolos por su nombre. Escribir

> precio_fruta <- c(banana = 50, pera = 30, naranja = 70) y

luego recuperar el valor de las naranjas con

> precio_fruta['naranja'].

Una vez creado un vector puede ser necesario modificarlo eliminando o agregando elementos o de otras maneras. Escribir

> vect <- c(5, 7, 11, 26)

Se desea remplazar 11 por 999.

> vect[3] <- 999. Verificar el cambio escribiendo

> vect

También se puede agregar un elemento generando una nueva posición donde alojarlo.

> vect[5] <- 111

Para eliminar un elemento se lo identifica con el signo -.

> vect <- vect[-3]

Un vector puede ser objeto de diversas operaciones, como se muestra a continuación de modo expeditivo. El archivo mtcars se distribuye con la instalación de R y se carga con sólo escribir el nombre. Escribir

> mtcars

en la consola aparece todo el archivo; probablemente deban ir hacia arriba para encontrar los encabezamientos. Escribir

> mtcars$mpg

Más adelante se verá el uso del signo $ en este contexto. Lo que interesa aquí es que este comando extrae la columna mpg del archivo mtcars, es decir, ha extraido un vector. Asignar el vector a una variable, u objeto

> v <- mtcars$mpg

Ejecutar las siguientes operaciones sobre el vector v. Los ';' permiten colocar varias operaciones en un mismo renglón

> max(v); min(v); mean(v); sd(v); length(v)

Estos comandos devuelven números únicos: los valores extremos, el promedio y la desviación típica y el número de elementos que contiene.

En cambio

> cumsum(v) devuelve la suma acumulativa de los elementos en v

Un ejemplo que puede sorprender. Se trata de determinar cuales elementos del vector v = mtcars$mpg son mayores que 20

> v > 20

Probablemente se espera que devuelva números, pero el resultado en un vector lógico que indica con TRUE la posición de aquellos elementos que cumplen con la condición y con FALSE los que no la cumplen. Para recuperar los elementos numéricos hay que usar los elementos lógicos como índices

> v2 <- v[v>20]

Se tiene ahora los elementos numéricos pero sin ordenar. Hacer

> sort(v2) para ordenarlos.

Otras operaciones con vectores se ven a lo largo del curso.

1.1.2. Operaciones matemáticas

Se ejemplifican varias operaciones algebraicas con vectores. Se trabaja con sumas y promedios, cálculo de distancia euclidiana y geodésica, y productos de vectores. El producto de un vector por un número resulta en un vector cuyos elementos están multiplicados por ese número. Se distingue producto escalar (o producto punto), útil para calcular ángulos y distancias, el producto vectorial calcula áreas de paralelogramos definidos por dos vectores, el producto mixto permite calcular volúmenes.

Ejemplo A – Suma y promedios de vectores

Se evalúa la contaminación por botellas de plástico en una zona litoral. El muestreo se realiza en cinco sectores en tres años sucesivos; las botellas muestreadas se retiran del sitio para no ser contadas nuevamente. La Tabla 1.1 muestra los resultados del muestreo. Completar los datos faltantes en la Tabla 1.1 con las ayudas que se dan abajo

Se crean los vectores

> a17 <- c(5,14,0,30,0)

para el año 2017.

Sitio	Año 2017	Año 2018	Año 2019	Total
1	5	15	18	
2	14	24	22	
3	0	0	0	
4	30	32	33	
5	0	8	9	
Suma parcial				
Promedio				

Tabla 1.1

```
> a18 <- c(15,24,0,32,8)
> a19 <- c(18,22,0,33,9)
```

Se calcula el total

```
> total <- a17 + a18 + a19
```

Las sumas parciales

```
> sum(a17) y así.
```

Los promedios parciales

```
> mean(a17) y así.
```

Ejemplo B - Distancia euclidiana entre dos puntos

La Tabla1.2 muestra coordenadas en proyección POSGAR 98, faja 3, para varias ciudades del noroeste de Argentina. Se trata de calcular las distancias entre ellas. Organizar los datos en estructuras de vectores: ciudad <- c(X,Y). Por ejemplo,

CiudadXY	X	Y
Santiago	3667285	6924138
Tucuman	3581403	7031878
La Rioja	3417762	6751041
Salta	3562763	7256922
Jujuy	3575588	7325100
Catamarca	3518297	6850478

Tabla 1.2

```
Santiago <- c(3580472,6924138)
Tucuman = c(3581403,7031878)
LaRioja = c(3417762,6751041)
Salta = c(3562763,7256922)
Jujuy = c(3575588,7325100)
Catamarca = c(3518297,6850478)
```

El paquete LearnGeom contiene funciones para facilitar operaciones matemáticas en geometría plana, incluyendo recursos gráficos sencillos.

```
> install.packages('LearnGeom')
> library(LearnGeom)
```

La función DistancePoints() en el paquete LearnGeom devuelve la distancia entre dos puntos.

```
> DistancePoints(Santiago,Tucuman)
```

devuelve la distancia euclidiana, es decir, horizontal, en metros entre esas dos ciudades, 107744 metros.

Ejemplo C - Distancia geodésica entre dos puntos

El producto escalar de dos vectores permite determinar la distancia entre Buenos Aires y el polo Sur considerando la curvatura de la Tierra. La posición de un punto P sobre la superficie terrestre está dada por la latitud, lat, y la longitud, lon, y en términos de trigonometría esférica la posición de P es

P <- (r*cos lat * cos lon, r*sin lat * cos lon, r*sin lat).

Ver Figura1.2.

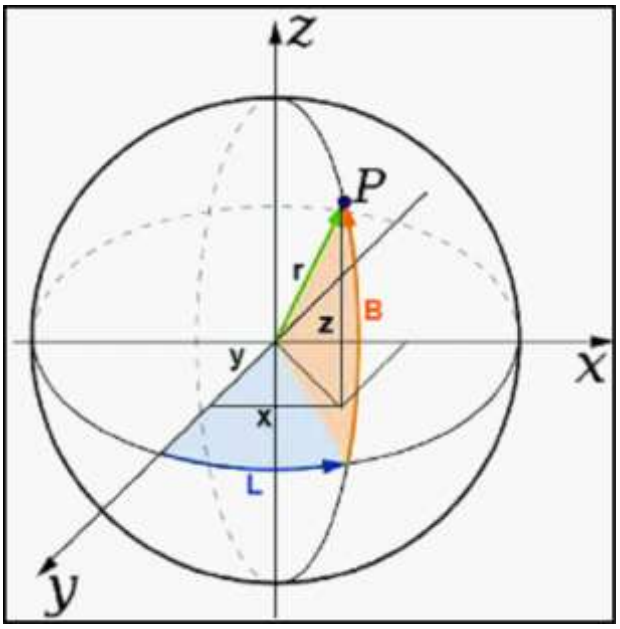

Figura 1.2: *Distancia geodésica.*

Polo Sur latitud l2 <- 90ºS, longitud z2 <- 58.38ºO Buenos Aires latitud l1 <- 34.61ºS, longitud z1 = 58.38ºO
Las posiciones de estos dos puntos en la esfera están dadas por

> ABA = c(cos(l1)*cos(z1),cos(l1)*sin(z1),sin(l1))

> APS = c(cos(l2)*cos(z2),cos(l2)*sin(z2),sin(l2))

Tomando como origen el centro de la Tierra, O, se definen dos vectores: OABA y OAPS, cuyo producto escalar determina el arco geodésico que los une y del arco se obtiene el ángulo entre ellos en radianes.

> ang <- acos(sum(ABA * APS))

y finalmente la distancia geodésica resulta de multiplicar el ángulo por el radio terrestre, r <- 6371 km

> dist <- ang * r <- 7381 km.

Ejemplo D - Proyección de un vector

Dados los vectores
u = c(2,0); v = c(3,-4)
se trata de calcular la magnitud del vector P resultante de la proyección de $\dot{u}$ sobre $\dot{v}$ (Figura1.3). La ecuación a resolver es:

$$\dot{P} < -\frac{\dot{u}\,\dot{v}}{||\dot{v}||^2}\,\dot{v}$$

La ecuación involucra el producto escalar, o producto punto (’dot product’) u x v, dividido por el módulo al cuadrado, y finalmente multiplicado por el vector sobre el cual se hace la proyección.

Nota:

En la ejecución de este ejercicio se emplean las funciones dot() y Norm(), del paquete pracma. El numerador se calcula con dot(u,v), el producto escalar o producto punto. La ecuación expandida es:
dot(a,b) <- a1b1 + a2b2 + a3b3
Otro modo de expresarlo:
producto punto u̇ x v̇ <- modulo u̇ x modulo v̇ x cos(ángulo entre u y v). Algunas equivalencias:
dot(a,b) <- a dot() opera sobre vectores o columnas de una matriz, y devuelve un escalar.

La sintaxis de Norm(x, p = 2), p = 2 por defecto y devuelve un escalar que representa la magnitud, o el largo euclidiano, del vector x. Por ejemplo, Norm(c(3,4)) devuelve 5, por Pitagoras. Norm() tiene equivalencias:

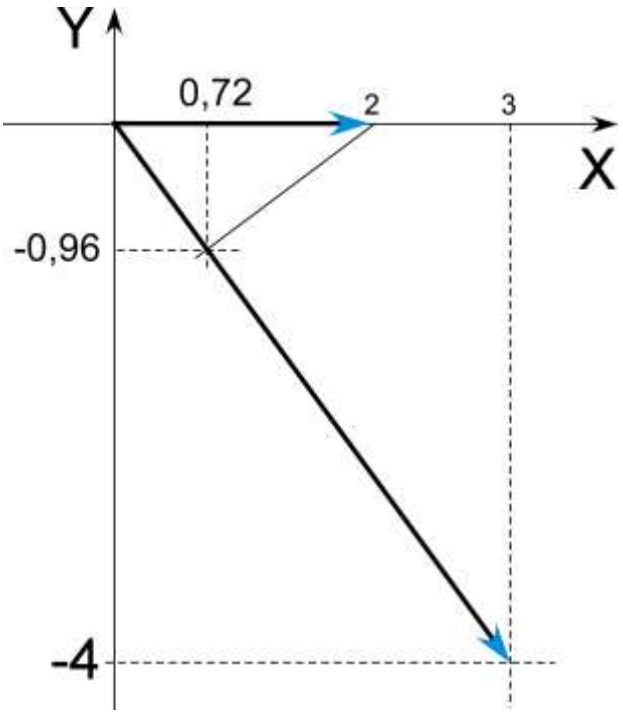

Figura 1.3: *Proyección de vectores.*

*Norm(x) <- sqrt(sum(x * x)).*

El numerador se obtiene con

> dot(u,v) <- 6

y el denominador con

> Norm(v)^2 <- 25

donde Norm(), con mayúscula, calcula el módulo, o magnitud del vector. Finalmente,

> 6/25 * $\dot{v}$ devuelve x <- 0.72 y y <- 0.96, las coordenadas del vector proyectado.

Ejemplo E - Cálculo del ángulo entre dos vectores

Se trata de obtener el ángulo entre los vectores $\dot{u}$ y $\dot{v}$ del Ejercicio D (Figura 1.3). La ecuación es:
angulo <- acos(sum(u*v) / (sqrt(sum(u * u)) * sqrt(sum(v * v)))) donde sum(u*v) <- dot(u,v) y sqrt(sum(u * u) <- Norm(u).
El resultado es en radianes; convirtiendo a grados da 53º.

Ejemplo F - Producto mixto

Calcular el volumen del paralelepípedo delimitado por tres puntos en el espacio cartesiano

> a = c(0,3,0); b = c(5,0,0); c = c(5,3,4) (Figura1.4).

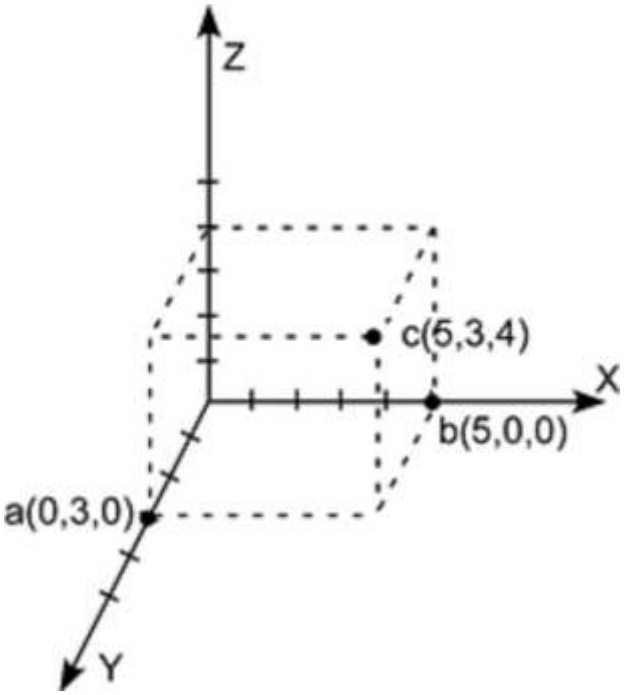

Figura 1.4: *Producto mixto.*

El cálculo del volumen involucra un producto escalar seguido por uno vectorial

> abs(dot(a,(cross(b,c))))

se toma el valor absoluto pues puede dar negativo.

Ejemplo G - Momentos de fuerza

El producto cruzado de dos vectores resulta en un vector perpendicular al plano formado por aquellos. En R el producto cruzadode vectores se calcula con cross(). Hacer

> cross(c(1,0,0),c(0,1,0)) devuelve 0 0 1

El ejercicio debe calcular la fuerza que se ejerce sobre la bisagra de un estante (Figura1.5). El vector fuerza tiene componentes
Fx <- 0 N Fy
<- 157 N

Fz <- 118 N
f <- c(0,157,118)
y se aplica a una distancia de 0,4 metros de la bisagra.

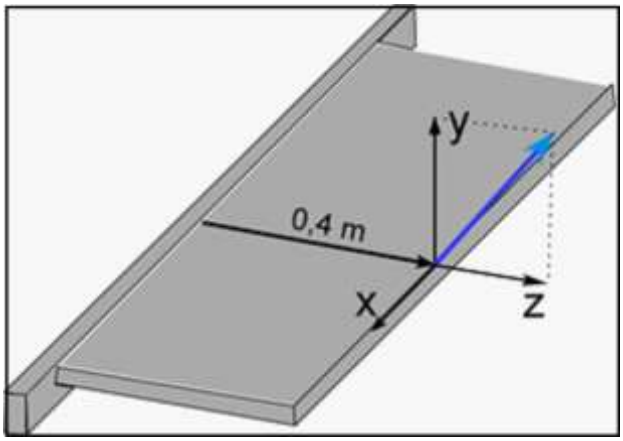

Figura 1.5: *Momento de fuerza.*

Describir el vector palanca r r
<- c(0,0,0.4)
Calcular el producto cruzado
cross(r,f)
devuelve -62.8 N

1.1.3. Reciclado

Las operaciones entre vectores exigen que tengan igual número de elementos. Pero ¿qué ocurre si esta condición no se da? Para explorar una respuesta crear los vectores A = c(1,2,3) y B = c(4,5,6,7,8) y sumarlos. Se obtiene la respuesta 5 7 9 8 10 pero acompañada por un mensaje explicando que el número de elementos en ambos no son múltiplos. Pero veamos qué hizo R para obtener un resultado. Los elementos del vector A se sumaron uno a uno a los tres primeros elementos del vector B. Para cubrir los restantes dos dígitos en B R repitió los primeros dos dígitos en A, dando 8 y 10. Es como transformar A en A = c(1,2,3,1,2). Es decir, los elementos del vector más corto se reutilizan afin de completar la operación demandada. Este comportamiento se denomina reciclado. Prueben agregando un número a B para que sea múltiplo de A.

1.2. Atributos

Al describir propiedades de vectores se mencionó que poseen atributos, como, por ejemplo, name. Más aun, al inicio del capítulo se dijo que todos los objetos en R poseen atributos, los cuales ayudan a caracterizar los objetos. Es conveniente, entonces, presentar los atributos formalmente.

Los tres atributos más importantes son los que describen los nombres de los elementos: names, la dimensión de la estructura: dim, y la clase de la estructura, class. names contiene etiquetas para los elementos de un vector o una lista; dim se emplea en la creación de matrices y arrays. De hecho, como se ve enseguida, matrices y arrays son vectores con atributo dim. El atributo class devuelve un vector con cadenas de caracteres que identifica las clases que un objeto hereda.

Para recuperar un listado de atributos de un objeto se recurre a attributes(objeto). Escribir

> attributes(mtcars)

devuelve el contenido de los atributos names, row.names y class.

Si se desea revisar solamente uno de los atributos se usa attr. Escribir

> attr(mtcars, 'names')

devuelve los nombres de las columnas.

Los atributos no son estáticos. Es posible modificar atributos y también asignarlos si faltasen. Por ejemplo, la sentencia names <- nombre, permite asignar un nombre de atributo. Más adelante se verá cómo recurriendo a los atributos colnames y names se puede modificar estructuras.

1.2.1. Ejemplos

Dado
b <- c(1, 2, 3), names(b) devuelve NULL, pero si se asignan nombres a las variables

> b <- c('uno' = 1,'dos' = 2,'tres' = 3) names(b) devuelve los nombres.

Además de rescatar nombres dados previamente, names permite asignar nombres. Hacer

> f <- c(4, 5, 6) y names(f) <- c('cuatro', 'cinco', 'seis')

Verificar que se incorporaron los nombres al vector.

> dim(mtcars) devuelve 32 11, o sea, 32 filas y 11 columnas

> dimnames(mtcars) devuelve los nombres de lascolumnas.

Un último ejemplo

> class(mtcars) devuelve 'data.frame', una cadena de caracteres.

1.3. Matrices

Una matriz es un arreglo bidimensional de valores dispuestos en filas y en columnas. Una matriz hereda la propiedad atómica de los vectores, por lo cual sólo puede contener elementos de idéntica clase. Una matriz se puede crear empleando tres funciones: matrix, cbind y rbind.

1.3.1. matrix()

La función matrix() crea una matriz a partir de un único vector definiendo las dimensiones. La función matrix() tiene la sintaxis por defecto: matrix(data = NA, nrow = 1, ncol = 1, byrow = FALSE, dimnames = NULL) creando una matriz de un elemento NA, ordenada por columnas y sin nombres de filas ni columnas.

De estos parámetros data y nrow o ncol, son obligatorios; dado nrow, por ejemplo, R deduce ncol. Los siguientes ejemplos muestran el uso de matrix() generando matrices con 10 elementos numéricos. Por defecto matrix() organiza los datos en columnas, o sea, completa una columna antes de pasar a a otra. Escribir

> matrix(1:10, nrow = 5) y comparar con

> matrix(1:10, nrow = 2)

Pero se puede exigir que lo haga por filas

> matrix(1:10, nrow = 2, byrow = TRUE)

En una matriz, los números entre [ ] encabezando las filas y las columnas indican la posición de cada elemento en la matriz [fila,columna]; [,2] indica todas las filas de la columna 2, y [2,] indica todas las columnas de la fila 2.

El número de elementos en una matriz se obtiene con

length(matriz)

Una matriz creada con matrix() carece de encabezamientos de columnas o índices de filas. Pero se puede asignar ambos empleando las funciones rownames() y colnamens().

rownames(matriz) <- c('col1', 'col2')

colnames(matriz) <- c('ind1','ind2')

1.3.2. cbind(), rbind()

Las funciones cbind() y rbind() crean matrices a partir de vectores. cbind() los une a modo de columnas y rbind() lo hace como filas, como muestran los ejemplos abajo.

Ejemplos

Crear una matriz uniendo vectores en columnas

mat_col <- cbind(1:5, c(1.7, 1.8, 1.9, 1.5, 1.1), c(55, 66, 44, 34, 59))

Crear una matriz uniendo vectores en filas

mat_fila <- rbind(1:5, c(1.7, 1.8, 1.9, 1.5, 1.1), c(55, 66, 44, 34, 59))

Notar que en ambos casos se operó con los mismos vectores.

Las columnas de una matriz pueden tener nombres dados por el usuario. Crear los vectores

> Edad <- c(28, 30, 33, 29, 23) y

> Peso <- c(55, 58, 57, 60,63)

Escribir

> matriz_nombre <- cbind(Edad, Peso)

Los nombres de las columnas se pueden agregar con posterioridad a la creación de la matriz, y también se pueden modificar, mediante la función colnames().

> colnames(matriz_nombre) <- c('Edades', 'Pesos')

De modo similar se opera con rownames() para asignar nombres a las filas.

1.3.3. Acceder y extraer elementos

Los métodos de acceso a los elementos individuales en una matriz son si- milares a los vistos para vectores, pero con doble subíndice. Empleando matriz_nombre definida previamente:

> matriz_nombre[3,2] devuelve 57

Si el espacio de fila o de columna se deja vacío R interpreta que se desea acceder a todos los elementos. Por ejemplo

> matriz_nombre[,1] devuelve todas las edades. Probar con matriz_nombre3,].

> matriz_nombre[3,2] <- 999 remplaza 57 por 999

> matriz_nombre[3,] <- c(00, 111) remplaza los valores en la fila 3

> matriz_nombre[,-1] elimina todos los elementos de la columna 1.

1.3.4. Operaciones matemáticas

El producto de una matriz por un número resulta en cada elemento de la matriz multiplicado por ese número. Los ejemplos que siguen muestran otros casos.

Ejemplo A - Optimización de la asignación de recursos

Este ejemplo aplica programación lineal (PL), una técnica de optimización de la asignación de recursos. El problema se plantea así: Dada una demanda a satisfacer y varias opciones para satisfacerla, cada opción con un costo de implementación, por lo tanto es de interés hallar la manera más eficiente de satisfacer dicha demanda, y para ello se apela a la técnica de asignación de recursos.

Sujeto	Limpiar Baño	Limpiar Pisos	Limpiar Ventanas
Pablo	$20	$30	$30
David	$30	$20	$30
Cristian	$30	$30	$20

Tabla 1.3

En un comercio hay que limpiar baños, pisos y ventanas. La tabla resume las tareas y el personal a asignar a ellas, indicando el precio que cada persona desea cobrar por ejecutarla (Tabla1.3). El gerente debe asignar el personal de modo que las tareas se ejecuten con menor costo. Se aplica el algoritmo de optimización conocido como método húngaro. La tabla se encuentra en la carpeta **Datos** como Asigna.txt.

En este ejemplo se utiliza la función lp.assign() incluida en el paquete lpSolve, un conjunto de algoritmos para programación lineal. lp.assign() opera sobre una matriz donde las filas representan las restricciones y las columnas las variables, y determina el costo de asignar una fuente a un destino. Incorporar lp.assign() con

> install.packages('lpSolve')

> library(lpSolve), o desde el GUI de RStudio.

Ingresar la tabla Asigna.txt en RStudio con
File>Import Dataset>From Text (readr).
La función de asignación de recursos lp.assign() requiere una matriz numérica, por lo cual hay que eliminar las cadenas de caracteres en la columna Sujeto. Una manera de hacerlo es reescribiendo la tabla sin esa columna

> asig <- Asigna[,2:4]

Seguidamente se convierte la tabla en una matriz con

> mat_asig <- data.matrix(asig). Verificar que se tiene una matriz numérica con

> mat_asig

Ahora sí se puede ejecutar

> lp.assign(mat_asig) que devuelve $60 como el menor costo posible para ejecutar las tareas con ese personal.

Ejemplo B - Sistemas de ecuaciones

Se trata de resolver el sistema de ecuaciones

$$x_1 + 3x_2 = 7$$

$$2x_1 + 4x_2 = 10$$

En forma matricial

$$\begin{matrix} 1 & 3 \\ 2 & 4 \end{matrix}$$

En forma compacta: A^{-1}b = x donde A^{-1} es la inversa de A. Para invertir una matriz se usa la función solve(), del paquete base.

Los datos son:
A <- matrix(c(1,2,3,4), ncol = 2)
b <- c(7, 10)

y la respuesta se obtiene con: x
<- solve(A) %*% b
que devuelve

$$x = \begin{matrix} \Sigma & \Sigma \\ 1 & \\ 2 & \end{matrix} \qquad (1.2)$$

Comprobar que ésta es la solución insertando x en las ecuaciones 1.1.

1.4. Arrays

Si el arreglo de vectores tiene tres o más dimensiones constituye un array. Un array hereda la propiedad atómica de vector. La sintaxis por defecto es: array(data = NA, dim = length(data), dimnames = NULL).

Los dos primeros parámetros son obligatorios. En tres dimensiones la estructura es: número de filas, número de columnas, número de capas o de matrices. Las dos primeras dimensiones configuran una matriz y la tercera corresponde al número de matrices. dimnames = refiere a una lista con los nombres de cada dimensión; por defecto R asigna números enteros a filas y columnas, como se muestra en los ejemplos que siguen y en el script Arrays.Rmd en scripts Estructuras.

Ejemplo A

Hacer

```
> array(1:18, dim = c(3, 3, 2)).
```

La salida se presenta como dos matrices, una por cada capa representada por los índices [, , 1] y [, , 2]. Verificar que los valores se distribuyen llenando primero las columnas.

Ejemplo B

Dados los vectores vect1 <- c(10, 20, 30, 40) y vect2 <- c(50, 60, 70, 80, 90, 100) el array B con dimensiones 3 3 2 se crea con

```
> B <- array(c(vect1, vect2), dim = c(3, 3, 2))
```

Ejemplo C

Asignar nombres a las dimensiones del vector B <- array(c(vect1, vect2), dim = c(3, 3, 2)) Los nombres son:

```
> r <- c('f1','f2','f3')
> c <- c('c1','c2','c3')
> m <- c('m1','m2')
> B_nom <- array(B, dim = c(3,3,2), dimnames = list(r,c,m))
> B_nom
```

Ejemplo D

La tarea consiste en crear un array a partir de vectores con datos del relevamiento de contaminación por botellas de plástico y latas de aluminio inicialmente presentado en el acápite 1.1.2. Abajo se dan los vectores con datos para botellas y latas por año y por sitio.

```
> a17_bot = c(5,14,0,30,0)
> a18_bot = c(15,24,0,32,8)
> a19_bot = c(18,22,0,33,9)
> a17_lat = c(3,3,5,1,2)
> a18_lat = c(5,6,2,7,3)
> a19_lat = c(8,9,6,11,13)
```

Se muestra la matriz para el Año 2017 (Tabla1.4).

Botellas	Latas
5	3
14	3
0	5
30	1
0	2

Tabla 1.4

Con estos datos crear un array de tres dimensiones.

a) Convertirlas a matrices a17_mat <- as.matrix(a17) repetir para a18 y a19.
b) Crear el array

array_cont <- array(c(a17_mat,a18_mat,a19_mat),c(5,2,3))
donde el primer vector reúne las tres matrices y el segundo da las dimensiones del array. El resultado es un array de tres dimensiones. Guardarlo como array_cont.

1.4.1. Acceder y extraer elementos

El acceso a un elemento en particular se hace con índices

> B_nom[1,1,1] devuelve 10

Ejemplo E

Extraer los elementos 40 de m1 y 30 de m2

> B_nom[1,2,]

Ejemplo F

Eliminar la fila f2 de m2.

> B_nom[-2,,2]

1.4.2. Operaciones con arrays

Las operaciones matemáticas con arrays son en realidad operaciones matriciales. Se dan ejemplos empleando el array array_cont, creado en el Ejemplo D arriba.

> array_cont * 2 duplica cada uno de los elementos en array_cont Probar con

> array_cont + array_cont

> array_cont * array_cont

> array_cont / array_cont

> array_cont%*% array_cont (producto matricial)

> array_cont * coeficiente (multiplicación por un vector).

Capítulo 2

Estructuras no atómicas

Una característica común a muchos conjuntos de datos es que contienen datos de más de un tipo. Un informe con resultados de pruebas toxicológicas puede tener el nombre de los pacientes, un carácter identificador de la prueba y números reales con los resultados. Es decir, conforma una estructura no atómica. ¿Cómo operar con datos de tipos diferentes? R ofrece dos estructuras de datos especialmente útiles para estos casos. En este capítulo se introducen las listas y los marcos de datos, estructuras que permiten combinar tipos de datos.

2.1. Listas

Las listas son estructuras semejantes a vectores pero que no son atómicas, es decir, pueden combinar tipos de datos. Una lista se crea con la función list() y tiene la siguiente estructura:
list(objeto1, objeto2, ...) donde objeto puede ser cualquier tipo de dato, incluyendo otra lista. Escribir

> Mi_lista <- list(pi, c(TRUE, FALSE), 'Mi lista', 2:10)

Esta lista comprende cuatro componentes, el número pi, un vector con elementos lógicos, una cadena de caracteres y un vector numérico.

Se puede asignar nombres a los componentes de una lista.

> Mi_Lista <- list(num_pi = pi, Bool = c(TRUE, FALSE), cadena = 'Mi lista', vector = 2:10).

Otro ejemplo

> Otra_lista <- list(c('Ene','Feb','Mar'), matrix(c(1,2,3,4,-1,9), nrow = 2),list('Rojo',12.3))

Ver script ListasRmd en scripts Estructuras.

> names(Otra_lista) <- c('Mes', 'Matriz', 'Misc')

> Otra_lista

Las listas son estructuras recursivas, es decir, una lista puede contener una o más listas. Escribir

> lista_recursiva = list(list(a = 5,b = 8),list(c = 'yo',d = 20))

2.1.1. Acceder y extraer elementos

El modo de acceso a los componentes de una lista es algo diferente que para vectores. La ejecución de

> Mi_lista

devuelve cada componente identificado por un entero entre [[ ]] y el identificador de orden para cada componente entre [ ]. Los dobles corchetes rectos acceden a los elementos individuales. Por ejemplo, para acceder al vector lógico completo hacer

> Mi_lista[[2]] devuelve el vector lógico y para acceder a uno de sus elementos

> Mi_lista[[2]][1] devuelve el primer elemento de dicho vector. En cambio

> Mi_Lista[2] $Bool

[1] TRUE FALSE

devuelve el vector lógico completo. En otros términos, los corchetes simples devuelven una tajada (slice) de la lista. Como ejercicio para comprender la diferencia guarden el resultado de cada caso como vector e intenten acceder a un elemento en particular, por ejemplo FALSE.

Si los componentes tienen nombres, entonces se usa el operador $ para relacionar la lista con el componente.

> Mi_lista$num_pi

Escribir

> Mi_lista$cadena <- 'Tu_lista' para remplazar 'Mi_lista' por 'Tu_lista'.

Acceder por nombre

> Otra_lista$Matriz

Los nombres remplazan los [[]] y [].

Veamos el caso de la lista_recursiva

> lista_recursiva = list(list(a = 5,b = 8),list(c = 'yo',d = 20))

Escribir

>lista_recursiva[[1]]

El primer par de corchetes accede a la lista exterior y el segundo a la interior que ocupa el primer lugar,

$a
[1] 5

$b
[1] 8

Con lista_recursiva[[2]] devuelve

$c

[1] "yo"

$d

[1] 20

2.1.2. Operaciones con listas

Operar con estructuras atómicas numéricas no ofrece problemas. Pero ¿cómo se opera con listas? Las operaciones matemáticas con listas están restringidas a los elementos que no son caracteres o lógicos. Escribir

> lista <- list('perro', 'gato', 5, 6) Se desea multiplicar los números 5 y 6. Se prueba con

> lista[3] devuelve 5 y lista[4] devuelve 6, por lo que se presume que

> lista[3] * lista[4] debiera dar el producto, pero da error al aplicar un operador binario a elementos no numéricos. La falla reside en que no se apuntó al elemento correctamente. Para verificar ejecutar

> lista[3]
[[1]]
[1] 5

Notar que el número 5 quedó en segundo rango. Escribir

> lista[[3]] * lista[[4]] y dará 30.

Las operaciones matemáticas con listas pueden hacerse separando los componentes por sus nombres. Por ejemplo

> Mi_lista$num_pi * 5^2 donde 5 es el radio de un círculo.

Repasar el acceso a los elementos de una lista. Si los elementos de la lista están agrupados se procede de modo diferente. Escribir

> lista2 <- list(a = c(5, 6), b = c('perro', 'gato')) comprende dos vectores.

> lista2$a extrae 5 y 6, y para multiplicarlos

> lista2$a[1] * lista2$a[2]

Anidar listas; incluir listas en una lista

> lista2 <- list('Lun','Mar','Mie', 'Jue', 'Vie')

```
> lista1 <- list(1,2,3,4,5)
> lista_unida <- list(lista2,lista1)
> lista_unida
```

Ejercicios optativos

1 - Crear una lista que incluya su nombre, su edad, su peso con decimales, su sexo y un valor lógico indicando si su pelo es rubio. 2 - Crear una lista que incluya dos listas. 3 – Dada la lista lista_unida, creada arriba. Escribir el comando que devuelve el elemento 3. 4 – Reunir los componentes de lista1 y lista2 en una nueva lista asignándoles nombres dias y num. La nueva estructura debe responder al comando

> lista$ devolviendo un panel con las opciones dias y num.

2.2. Dataframes y tibbles

El dataframe es la estructura de batalla en R, incluído en la instalación base. Un dataframe se asemeja a una matriz en cuanto a que tiene filas y columnas. Las filas pueden contener tipos disímiles de datos, como las listas, pero no así las columnas, las cuales deben ser atómicas. Las filas deben todas tener idéntico largo.

Y el tibble ¿qué es? En 2016 se propuso un sustituto, actualmente conocido como tibble, que ha ido remplazando al dataframe tradicional. Estructuralmente es similar al dataframe. Un tibble se puede categorizar como un dataframe con mejoras. Las principales mejoras son las siguientes:

- Tibble escribe a consola la dimensión y el tipo de datos, los primeros 10 registros y las columnas que quepan en la pantalla. Esto tiene la ventaja de que no abarrota la pantalla pero la desventaja de que puede dejar fuera columnas de interés. Ensanchando la consola se agregan columnas.
- Tibbles no coloca etiquetas para las filas (rownames); esto facilita la compatibilidad con tablas creadas en SQL.
- Tibble no convierte automáticamente las variables categóricas a factores. En rigor, data.frame tampoco lo hace a partir de la versi'on de R 4.0.
- Un subconjunto tibble devuelve otro tibble, mientras que un dataframe puede devolver tanto un dataframe como un vector.

El ecosistema tidyverse, y por lo tanto el tibble, conforman el modo moderno de programar en R y se adopta en este curso.

La literatura en castellano suele traducir dataframe por marco de datos. En este curso aplicamos marco de datos tanto a dataframes como a tibbles, diferenciando entre ambos donde sea necesario.

Ver script MarcoDatos.Rmd en scripts Estructuras.

2.2.1. Crear dataframes y tibbles

La función data.frame() permite crear un dataframe a partir de vectores, listas o matrices. La sintaxis de la función data.frame() remeda la de cbind(), pues concatena vectores en forma de columnas. data.frame(nombre1 = col1, nombre2 = col2, nombre3 = col3, ...)

Ejemplo.

Se tiene una constante de caracteres, un vector de enteros y un vector lógico

```
> L3 <- LETTERS[1:3]
> num <- c(2, 3, 6)
> log <- c(TRUE, FALSE, TRUE)
```

Combinarlos en un dataframe

```
> data.frame(L3, num, log)
```

Resultado

```
> data.frame(L3,num,log)
  L3 num   log
1  A   2  TRUE
2  B   3 FALSE
3  C   6  TRUE
```

data.frame devuelve tres vectores con los nombres de las columnas, y por defecto números enteros designando las filas en una primera columna; esta columna no forma parte de los datos, es decir, no se puede operar con sus valores.

Convertimos los vectores a tibble

```
> tibble(L3,num,log)
# A tibble: 3 × 3
  L3      num log
  <chr> <dbl> <lgl>
1 A         2 TRUE
2 B         3 FALSE
3 C         6 TRUE
```

tibble devuelve también los tres vectores pero adicionalmente informa la dimensión de la estructura y los tipos de dato en cada vector. Por compatibilidad con los dataframes tradicionales, los tibbles incluyen etiquetas de filas, pero la moyor parte de los métodos en tidyverse los ignoran. Los números a la izquierda son orientativos, notar que están grisados.

¿A qué clase pertenecen un dataframe y un tibble?

> class(df) devuelve data.frame

> class(tb) devuelve tbl_df, tbl y data.frame.

Noten, pues, que un tibble hereda de un dataframe. De hecho tbl_df es una subclase de dataframe creada para agregar otras propiedades. En este curso trabajaremos mayormente con tibbles, puntualizando conflictos con los tradicionales dataframes donde sea necesario. Hacemos esta elección conscientes de que pueda generar alguna confusión cuando ustedes revisen textos y códigos algo antiguos, y también cuando recurran a conjuntos de datos preinstalados, como mtcars, que es un dataframe. En contraprestación, tidyverse, y los tibbles, son el futuro de la programación en R y es mejor incorporar este ecosistema al tiempo de aprender a programar en R. Al inicializar variables con dataframes les asignaremos el sufijo_df y a tibbles _tb.

Cargamos el conjunto de datos mtcars y lo abrimos en pantalla

```
> mtcars_df <- mtcars
> mtcars_df
```

Imprime los identificadores de filas y las columnas con datos. Imprime todas las columnas apilando las salidas.

Seguidamente lo convertimos a tibble.

```
> mtcars_tb <- as_tibble(mtcars)
```

```
> mtcars_tb
```

omite los identificadores de filas. Noten que no imprime en pantalla todas las columnas y al pie indica cuales se excluyeron.

2.2.2. Acceder y extraer elementos

El procedimiento para acceder a un elemento en particular, o a un subconjunto de elementos, en tibbles, es similar al visto para vectores y matrices.

```
> mtcars_tb [,3]
```

devuelve los elementos en todas las filas de la tercera columna,

```
> mtcars_tb [, 'cyl']
```

o también usando el nombre de la columna y un índice de posición

```
> mtcars_tb $mpg[5]
```

O emplear la función subset(), que opera sobre vectores y matrices para extraer una porción de los elementos. Por ejemplo

```
> subset(mtcars_tb, mpg > 20) o
> subset(mtcars_tb, mpg > 25 & mpg < 30) donde & implica Y lógico.
```

****

Nota:
En el módulo 1 presentamos el operador %>%, o tubo ('pipe'), Tubo concatena comandos pasando el resultado de uno al siguiente, y hace un código más legible. El siguiente ejemplo muestra porqué. Se tiene el vector x <- c(1, 4, 6, 8) y se trata de obtener el promedio de la raíz cuadrada de x redondeada a dos decimales. Tradicionalmente el comando es

```
> y <- round(mean(sqrt(log(x))), 2)
```

La sucesión de operaciones es a la inversa de como está escrito el comando; la primera está en el núcleo. Empleando tubos el comando queda así

```
>  y <- x %>% log() %>% sqrt() %>% mean() %>% round(2)
```

La sucesión de operaciones está claramente evidenciada de izquierda a derecha.

El operador tubo fue introducido en el paquete magrittr en 2014 y el reciente combo-paquete tidyverse lo incluye y propicia su uso.

Retomemos la última sentencia.
> mtcars_tb %>% subset(mpg > 25 & mpg < 30)
Esta sintaxis refleja con mayor claridad el orden de las operaciones. Primero se cargan los datos y luego se aplica la función con sus argumentos.

Capítulo 3

Factores y funciones

Quedan dos importantes estructuras sin presentar: factor y function. Son estructuras compuestas que suelen incluir diversos tipos de datos y cumplen funciones muy relevantes.

3.1. Factores

La estructura factor es la más peculiar en R y está particularmente adaptada para el trabajo con datos ordinales y categóricos, particularmente con conjuntos que resultan de la repetición de un número limitado de elementos. Los datos ordinales representan un orden cualitativo, como alto – mediano – bajo, mientras que los categóricos no lo hacen, por ejemplo, azul – verde –rojo.

Supongamos un experimento que registra el color de ojo de 50 individuos como azul, marrón y verde. El vector contendrá 50 elementos que son reiteración de tres elementos base, los tres colores registrados. Al crear una estructura factor, R convierte los nombres de colores en números enteros al tiempo que registra los elementos que se repiten como cadenas, asignándolos al atributo levels. El ejemplo resultaría en 50 dígitos y tres levels: 'azul', 'verde', 'marron'.

La función factor() convierte un vector en un factor, como se muestra en el siguiente ejemplo.

3.1.1. Ejemplo con tipos ordinales

Un torno opera a tres velocidades: baja, media y alta y el registro de un experimento da lugar al vector:

```
> velocidad<-c('alta','baja','alta','media','alta','media','alta','baja','alta')
```

```
> fv <- factor(velocidad)
```

convierte el vector a una variable de factores

```
> fv
```

devuelve una estructura que semeja el vector original con el agregado del atributo Levels, que indica los elementos que se repiten. La semejanza con el vector original esconde dos cambios asociados a la conversión a factor. Uno es que las cadenas de caracteres son remplazadas por números enteros, y el otro es que la estructura es simplificada a un producto de cada level por el número de veces que se repite, ahorrando en espacio de almacenaje.

3.1.2. Ejemplo con tipos categóricos

Un experimento registra los colores de ojos de varios sujetos

> h <- c(’azul’, ’verde’,’azul’,’verde’,’amarillo’,’rojo’,’rojo’)

> hh <- factor(h) convierte el vector a factores, y

> hh devuelve como Levels los colores base.

Por defecto los Levels salen en orden alfabético, por ejemplo, alta, baja, media, lo cual no refleja el ordenamiento cualitativo. Para cambiar el orden se emplea el argumento levels =

> fv <- factor(velocidad, levels = c(’baja’, ’media’, ’alta’)

> fv

devuelve los Levels en el orden deseado.

De modo similar, el argumento labels <- permite dar o modificar los nombres de los Levels. En un experimento los sujetos debían describir un estímulo como fuerte o débil, pero para simplificar la tarea el analista optó por escribir 1 y 2, respectivamente, obteniendo d <- c(1,1,1,2,1,2,1,1,2,2,1,1,1).
Transformando el vector a factor se aprecia que tiene dos niveles, 1 y 2. En el informe, sin embargo, deben figurar los nombres correctos, entonces

> d.f <- factor(d, levels <- c(1, 2), labels <- c(’fuerte’, ’debil’))

Cuando el vector contiene elementos ordinales es recomendable emplear la función ordered(), que opera de manera similar a factor() pero ordena correctamente los levels. Escribir

> velocidad <- c(’alta’ , ’baja’ , ’alta’ , ’media’ ,’alta’ , ’media’, ’alta’ , ’baja’ , ’alta’)

> fo <- ordered(velocidad, levels = c(’baja’,’media’,’alta’))

> fo devuelve Levels: baja < media < alta

Dados los vectores

> altura <- c(132,151,162,139,166,147,122)

> peso <- c(48,49,66,53,67,52,40)

> genero <- c(’hombre’,’hombre’,’mujer’,’mujer’,’hombre’,’mujer’,’hombre’)

Crear un marco de datos

> md_tb <- tibble(altura, peso, genero)

Supongamos que la columna con cadenas, genero, debe ser convertida a factor. Primero genero una variable con los niveles

> gen <- c(‘hombre’,’mujer’)

> md_tb_fact <- factor(md_tb$genero, levels=gen)

Comprobar la conversión.

Acceder y extraer elementos

El acceso a los elementos de un factor sigue reglas similares que para vectores.

> md_tb_fact[3]

devuelve mujer.

Una ventaja de los factores se da en el almacenaje. Cada elemento único se guarda una sola vez y el número de veces que se repite se guarda como un

número entero. Pero la mayor aplicabilidad es en el trabajo con modelos estadísticos, debido a que las variables continuas son procesadas en R de modo diferente de las categóricas, como se verá más adelante al presentar modelos de regresión lineal.

3.2. Funciones

Una función es un conjunto de sentencias, claramente segregadas del resto del script, cuya ejecución da un resultado que puede ser utilizado por el script. Una función se compone de un nombre, de la palabra reservada function que asigna a ese nombre el carácter de función, de cero o más argumentos requeridos por la función, entre paréntesis, y de una o más sentencias que determinan el comportamiento de la función y el resultado de la ejecución, entre llaves.

La sintaxis es:

```
nombre_de_funcion <- function(listado de argumentos)
{
Código que ejecuta una acción
}
```

Al invocar una función se le pasan valores a los argumentos, siempre y cuando la función lo requiera. Ejemplo:

```
suma <- function(x, y) x + y
```

Invocar con:

```
suma(4, 6)
suma(x  4, y = 6)
suma( y = 6, x = 4)
```

Es decir, los valores se pueden pasar por posición o por nombre.

El resultado de la ejecución del código se almacena en nombre_de_funcion. Las funciones son objetos de la clase function.

Una virtud importante de las funciones es que son autocontenidas y por lo tanto pueden ser reutilizadas en otros scripts.

3.2.1. Argumentos por defecto

En las páginas previas se presentaron varias funciones que ya están instaladas en R, como por ejemplo ls() y matrix(). Los paréntesis vacíos no implican ausencia de argumentos, es la sintaxis del nombre elemental. Casi todas las funciones incluyen argumentos. Para revelarlos escribir el nombre de la función omitiendo los paréntesis. Por ejemplo

```
> ls
```

devuelve una primera línea con

```
function(name,pos=-1L,envir=as.environment(pos), all.names = FALSE, pattern, sorted = TRUE)
```

donde a la palabra reservada function le siguen los argumentos disponibles. Notar que todos los argumentos, excepto name, ya tienen asignados valores por defecto, y en el caso de name el código que sigue explicita que si el analista no le da un valor el programa le asigna uno. Por esto al escribir

```
> ls()
```

se obtiene un resultado a pesar de no llevar un argumento. Los argumentos pueden ser números, vectores, matrices o cadenas de caracteres.

Asimismo, una función puede tener un único argumento o varios, y más aun, uno de los argumentos puede estar predefinido, como se muestra en el siguiente ejemplo.

La función f tiene como argumentos un número real y un vector numérico que ya tiene asignado valores

```
f <- function(x, y = c(2,3,4))
{
xˆy
}
```

```
> f(3) devuelve 9 27 81.
```

El vector y establece los exponentes que afectan a x = 3.

Aprovechemos esta sencilla función para comprender el comportamiento.

Variante 1 - Se pasan valores a x pero también a y

```
> f(3,5)
```

de modo que y toma el valor 5. El resultado da 243 = 3 ˆ 5. Ocurre que el argumento predefinido y <- c(2,3,4) es una variable de alcance (’scope’) local, es decir, interno a la función, mientras que 5 es un valor global y tiene preeminencia. Las variables definidas dentro de una función tienen alcance limitado a dicha función.

Variante 2 - Se modifica f() asignando el resultado a una variable

```
f <- function(x, y = c(2,3,4))
{
a <- xˆy
}
```

Invocación con f(3), como inicialmente, no arroja resultado. Escribiendo con a tampoco, debido a que a tiene alcance local y el script no la puede ver. Es necesario escribir print(f(3)), o asignar la función a una variable

```
> var_temp <- f(3)
> var_temp
```

La Variante 2 indica que si el último comando en una función es de asignación el resultado no se imprime automáticamente en pantalla.

Una función puede devolver un único número (un vector unitario) o un vector con varios elementos Por ejemplo, dado x = c(2,-2,0,5), sum(x) devuelve 5 y cumsum(x) devuelve 2 0 0 5.

Nota:

Para ejecutar una función escribirla en el editor de RStudio, seleccionarla entera y pulsar Run para cargarla en memoria. Luego, en la Consola, escribir el comando de ejecución.

floor(), ceiling(), round(), signif()

Con frecuencia será conveniente reducir el número de dígitos decimales a la derecha de la coma en un número real. La modificación puede eliminarlos por completo dejando un número real o convirtiéndolo en uno entero. Se ejemplifica sucintamente con pi = 3.14.
floor(pi) da 3
ceiling(pi) da 4
trunc(pi) da 3
round(pi)

y por último,
signif(pi, 2) da 3.1, es decir, determina cuantos dígitos quedan.

args()

Claramente, es importante conocer cuales argumentos exige, o acepta una función para ejecutarla correctamente. Cuando uno mismo crea una función sabe que argumentos necesita. Con una función importada la naturaleza y cantidad de los argumentos puede no ser evidente. La función args() se usa para este fin
> args(sqrt) devuelve el argumento principal, x, y NULL, indicando que no requiere ni acepta otros argumentos. En cambio
> args(round) devuelve x pero también digits = 0, indicando que por defecto redondea sin decimales pero que eso se puede cambiar. Probar con round(pi, digits = 2)

3.2.2. Crear funciones

La instalación de R y los paquetes disponibles ofrecen decenas de funciones ya creadas. Adicionalmente el usuario puede crear funciones para resolver problemas específicos y almacenarlas para uso posterior. Si en un proyecto un mismo código aparece más de un par de veces hay que considerar conformar con él una función. De este modo no sólo se ahorran líneas de código sino también mejora la legibilidad del programa.

Ejemplo A

Un ejemplo sencillo es una función que eleve un número x a un exponente y.
> pot <- function(x, y) xˆy donde (x,y) son los argumentos y el término xˆy incluye el código. Para ejecutar pot() hay que pasar valores a los argumentos
> pot(2,5) que devuelve 32. El orden de los argumentos en pot(2, 5) debe concordar con el orden de x e y, a menos que se empleen nombres
> pot(y <- 5, x <- 2) también devuelve 32 aunque el orden de los argumentos se ha invertido. Si el resultado de pot() ha de ser usado en otro segmento del código asignarlo a una variable
> res <- pot(2,5)

Toda función comprende tres partes: body(), formals() y environment(). Hacer
> body(pot) devuelve xˆy

> formals(pot) devuelve $x $y

> environment(pot)

devuelve Global Environment porque fue creada globalmente.

Ejemplo B

Escribir una función que calcule los promedios de cada columna de la matriz mz

> mz <- matrix(1:32, c(4,8))

En una primera aproximación se crea la función

> f_prom <- function(x) mean(x) que realiza el cálculo en cada columna

> a <- f_prom(mz[,1]) !pero no se gana casi nada ya que hay que escribir la función ocho veces! Para evitar esto R ofrece una eficiente alternativa con la función apply()

> tot_prom <- apply(mz, MARGIN=2,f_prom)

apply() es una función que llama a otra función, en este caso f_prom(), y la aplica reiteradamente sobre el objeto mz. El segundo argumento, MARGIN, es un número entero, 1 o 2, que indica si la operación se hace sobre lasfilas, MARGIN = 1, sobre las columnas, MARGIN = 2, o sobre filas y columnas, MARGIN = c(1,2). apply() es la más básica de varias funciones similares que se verán al presentar operaciones de iteración.

Ejemplo C

La función bisiesto(), cuyo código se da abajo, determina si un año es bisiesto. Transcribirla al editor de scripts. Seleccionarla completa y pulsar en Run. Notarán que se reproduce en la consola. En la consola llamarla con bisiesto(1222), por ejemplo.

```
bisiesto = function(anio){
  if((anio %%      4)      ==      0) {
   if((anio %%     100)   ==      0) {
    if((anio %%    400)   ==      0) {
      print(paste(anio,"es bisiesto"))
    } else {
      print(paste(anio,"no es bisiesto"))
    }
  } else {
    print(paste(anio," es bisiesto"))
  }
  } else {
    print(paste(anio,"no es bisiesto"))
  }
}
```

El operador%% devuelve el módulo de la división. Si el año es exactamente divisible por 4 se prueba si es exactamente divisible por 100, si lo es se prueba si es exactamente divisible por 400, si lo es el año es bisiesto.

Si no tuvieren experiencia con bucles condicionales pueden regresar a este ejercicio luego de ver el capítulo Fundamentos de programación en R.

Nota:

Modo de ejecución. Pegar la función en el editor de RStudio. Seleccionarla con Ctrl-A. Pulsar Run. Luego pasar a la Consola y escribir

> bisiesto(2105)

return()

En ocasiones encontrarán funciones que incluyen el comando return(). La tarea de este comando es devolver el resultado al código para que sea utilizado. Los ejemplos dados muestran que la acción de retorno del valor calculado por la función se cumple sin necesidad de return(). Con ciertas funciones algo más complejas, sin embargo, return() es necesario. Un caso se da cuando la función devuelve varios resultados.

> fun <- function(x, y) z1 <- x + y z2 <- x* y

return(list(z1, z2))

Si se omite return() R no muestra el resultado. Notar que return() devuelve una lista con los resultados.

Ver scripts Alcance.Rmd, Funciones.Rmd, FuncionesAnonimas.Rmd y FuncionesContinuado.Rmd en scripts Estructuras.

Fin del Modulo 1