R Programming/Data types

Data types

Vectors are the simplest R objects, an ordered list of primitive R objects of a given type (e.g. real numbers, strings, logicals). Vectors are indexed by integers starting at 1. **Factors** are similar to vectors but where each element is categorical, i.e. one of a fixed number of possibilities (or *levels*). A **matrix** is like a vector but with a specific instruction for the layout such that it looks like a matrix, i.e. the elements are indexed by two integers, each starting at 1. **Arrays** are similar to matrices but can have more than 2 dimensions. A **list** is similar to a vector, but the elements need not all be of the same type. The elements of a list can be indexed either by integers or by named strings, i.e. an R list can be used to implement what is known in other languages as an "associative array", "hash table", "map" or "dictionary". A **dataframe** is like a matrix but does not assume that all columns have the same type. A dataframe is a list of variables/vectors of the same length. **Classes** define how **objects** of a certain type look like. **Classes** are attached to object as an **attribute**. All R objects have a class, a type and a dimension.

```
> class(object)
> typeof(object)
> dim(object)
```

Vectors

You can create a vector using the `c()` function which concatenates some elements. You can create a sequence using the `:` symbol or the `seq()` function. For instance `1:5` gives all the number between 1 and 5. The `seq()` function lets you specify the interval between the successive numbers. You can also repeat a pattern using the `rep()` function. You can also create a numeric vector of missing values using `numeric()`, a character vector of missing values using `character()` and a logical vector of missing values (ie `FALSE`) using `logical()`

```
> c(1,2,3,4,5)
[1] 1 2 3 4 5
> c("a","b","c","d","e")
[1] "a" "b" "c" "d" "e"
> c(T,F,T,F)
[1]  TRUE FALSE  TRUE FALSE

> 1:5
[1] 1 2 3 4 5
> 5:1
[1] 5 4 3 2 1
> seq(1,5)
[1] 1 2 3 4 5
> seq(1,5,by=.5)
[1] 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0
> rep(1,5)
[1] 1 1 1 1 1
> rep(1:2,5)
 [1] 1 2 1 2 1 2 1 2 1 2
> numeric(5)
[1] 0 0 0 0 0
> logical(5)
[1] FALSE FALSE FALSE FALSE FALSE
```

```
> character(5)
[1] "" "" "" "" ""
```

The `length()` computes the length of a vector. `last()` (**sfsmisc (http://cran.r-project.org/web/packages/sfsmisc/index.html)**) returns the last element of a vector but this can also be achieved simply without the need for an extra package.

```
x <- seq(1,5,by=.5)    # Create a sequence of number
x                      # Display this object
 [1] 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0
> length(x)            # Get length of object x
 [1] 9
> library(sfsmisc)
> last(x)              # Select the last element of x
 [1] 5.0
> x[length(x)]         # Select the last element wihout an extra package.
 [1] 5.0
```

Factors

`factor()` transforms a vector into a factor. A factor can also be ordered with the option `ordered=T` or the function `ordered()`. `levels()` returns the levels of a factor. `gl()` generates factors. `n` is the number of levels, `k` the number of repetition of each factor and `length` the total length of the factor. `labels` is optional and gives labels to each level.

Factors can be most easily thought of as categorical variables. An important function for factor analysis is the `table()` function, which offers a type of summary. When considering the types of statistical data (nominal, ordinal, interval and ratio), factors can be nominal, ordinal or interval. Nominal factors are categorical names, examples of which could be country names paired with some other information. An example of an ordinal factor would be a set of race times for a particular athlete paired with the athlete's finishing place (first, second, ...). When trying to summarize this factor, please see the example with ordinal examples below for an example on self-ordering your factors. Finally, an example of interval level factors would be age brackets such as "20 - 29", "30 - 39", etc. In general, R can automatically order numbers stored as factors appropriately but a programmer may use the same techniques with this type of data to order in the manner most appropriate to their application.

See also `is.factor()`, `as.factor()`, `is.ordered()` and `as.ordered()`.

```
> factor(c("yes","no","yes","maybe","maybe","no","maybe","no","no"))
[1] yes   no    yes   maybe maybe no    maybe no    no
Levels: maybe no yes
>
> factor(c("yes","no","yes","maybe","maybe","no","maybe","no","no"), ordered = T)
[1] yes   no    yes   maybe maybe no    maybe no    no
Levels: maybe < no < yes
>
> ordered(c("yes","no","yes","maybe","maybe","no","maybe","no","no"))
[1] yes   no    yes   maybe maybe no    maybe no    no
Levels: maybe < no < yes
>
> ordered(as.factor(c("First","Third","Second","Fifth","First","First","Third")),
+ levels = c("First","Second","Third","Fourth","Fifth"))
[1] First  Third  Second Fifth  First  First  Third
Levels: First < Second < Third < Fourth < Fifth
>
>  gl(n=2, k=2, length=10, labels = c("Male", "Female")) # generate factor levels
 [1] Male   Male   Female Female Male   Male   Female Female Male   Male
Levels: Male Female
```

Matrix

- If you want to create a new matrix, one way is to use the `matrix()` function. You have to enter a vector of data, the number of rows and/or columns and finally you can specify if you want R to read your vector by row or by column (the default option). Here are two examples.

```
> matrix(data = NA, nrow = 5, ncol = 5, byrow = T)
     [,1] [,2] [,3] [,4] [,5]
[1,]   NA   NA   NA   NA   NA
[2,]   NA   NA   NA   NA   NA
[3,]   NA   NA   NA   NA   NA
[4,]   NA   NA   NA   NA   NA
[5,]   NA   NA   NA   NA   NA
```

```
> matrix(data = 1:15, nrow = 5, ncol = 5, byrow = T)
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    2    3    4    5
[2,]    6    7    8    9   10
[3,]   11   12   13   14   15
[4,]    1    2    3    4    5
[5,]    6    7    8    9   10
```

- Functions `cbind()` and `rbind()` combine vectors into matrices in a *column* by *column* or *row* by *row* mode:

```
> v1 <- 1:5
> v2 <- 5:1
> v2
[1] 5 4 3 2 1
> cbind(v1,v2)
     v1 v2
[1,]  1  5
[2,]  2  4
[3,]  3  3
[4,]  4  2
[5,]  5  1

> rbind(v1,v2)
   [,1] [,2] [,3] [,4] [,5]
v1    1    2    3    4    5
v2    5    4    3    2    1
```

- The dimension of a matrix can be obtained using the `dim()` function. Alternatively `nrow()` and `ncol()` returns the number of rows and columns in a matrix:

```
> X <- matrix(data = 1:15, nrow = 5, ncol = 5, byrow = T)
> dim(X)
[1] 5 5
> nrow(X)
[1] 5
> ncol(X)
[1] 5
```

- Function `t()` transposes a matrix:

```
> t(X)
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    6   11    1    6
[2,]    2    7   12    2    7
[3,]    3    8   13    3    8
[4,]    4    9   14    4    9
[5,]    5   10   15    5   10
```

- Unlike data frames matrices must either be numeric or character in type:

```
> a=matrix(2,2,2)
> a
     [,1] [,2]
[1,]    2    2
[2,]    2    2
> a = rbind(a,c("A","A"))
> a
     [,1] [,2]
[1,] "2"  "2"
[2,] "2"  "2"
[3,] "A"  "A"
```

Arrays

An array is composed of n dimensions where each dimension is a vector of R objects of the same type. An array of one dimension of one element may be constructed as follows.

```
> x <- array(c(T,F),dim=c(1))
> print(x)
[1] TRUE
```

The array x was created with a single dimension (dim=c(1)) drawn from the vector of possible values c(T,F). A similar array, y, can be created with a single dimension and two values.

```
> y <- array(c(T,F),dim=c(2))
> print(y)
[1]  TRUE FALSE
```

A three dimensional array - 3 by 3 by 3 - may be created as follows.

```
> z <- array(1:27,dim=c(3,3,3))
> dim(z)
[1] 3 3 3
> print(z)
, , 1

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

, , 2

     [,1] [,2] [,3]
[1,]   10   13   16
[2,]   11   14   17
[3,]   12   15   18

, , 3

     [,1] [,2] [,3]
[1,]   19   22   25
[2,]   20   23   26
[3,]   21   24   27
```

R arrays are accessed in a manner similar to arrays in other languages: by integer index, starting at 1 (not 0). The following code shows how the third dimension of the 3 by 3 by 3 array can be accessed. The third dimension is a 3 by 3 array.

```
> z[,,3]
     [,1] [,2] [,3]
[1,]   19   22   25
[2,]   20   23   26
[3,]   21   24   27
```

Specifying two of the three dimensions returns an array on one dimension.

```
> z[,3,3]
[1] 25 26 27
```

Specifying three of three dimension returns an element of the 3 by 3 by 3 array.

```
> z[3,3,3]
[1] 27
```

More complex partitioning of array may be had.

```
> z[,c(2,3),c(2,3)]
, , 1

     [,1] [,2]
[1,]   13   16
[2,]   14   17
[3,]   15   18

, , 2

     [,1] [,2]
[1,]   22   25
[2,]   23   26
[3,]   24   27
```

Arrays need not be symmetric across all dimensions. The following code creates a pair of 3 by 3 arrays.

```
> w <- array(1:18,dim=c(3,3,2))
> print(w)
, , 1

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

, , 2

     [,1] [,2] [,3]
[1,]   10   13   16
[2,]   11   14   17
[3,]   12   15   18
```

Objects of the vectors composing the array must be of the same type, but they need not be numbers.

```
> u <- array(c(T,F),dim=c(3,3,2))
> print(u)
, , 1

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

, , 2

      [,1]  [,2]  [,3]
[1,] FALSE  TRUE FALSE
[2,]  TRUE FALSE  TRUE
[3,] FALSE  TRUE FALSE
```

Lists

A list is a collection of R objects. `list()` creates a list. `unlist()` transform a list into a vector. The objects in a list do not have to be of the same type or length.

```
> x <- c(1:4)
> y <- FALSE
> z <- matrix(c(1:4),nrow=2,ncol=2)
> myList <- list(x,y,z)
> myList
 [[1]]
[1] 1 2 3 4

 [[2]]
[1] FALSE

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

lists have very flexible methods for reference

- by index number:

```
> a <- list()
> a
list()
> a[[1]] = "A"
> a
[[1]]
[1] "A"

> a[[2]]="B"
> a
[[1]]
[1] "A"

[[2]]
[1] "B"
```

- By name:

```
> a
list()
> a$fruit = "Apple"
> a
$fruit
[1] "Apple"

> a$color = "green"
> a
$fruit
[1] "Apple"

$color
[1] "green"
```

- This can also be recursive and in combination

```
> a = list()
> a[[1]] = "house"
> a$park = "green's park"
> a
[[1]]
[1] "house"

$park
[1] "green's park"
```

```
> a$park = "green's park"
> a[[1]]$address = "1 main st."

> a
[[1]]
[[1]][[1]]
[1] "house"

[[1]]$address
[1] "1 main st."


$park
[1] "green's park"
```

Using the scoping rules in R one can also dynamically name and create list elements

```
>  a <- list()
>  n <- 1:10
>  fruit = paste("number of coconuts in bin",n)
> my.number = paste("I have",10:1,"coconuts")
> for (i in 1:10)a[fruit[i]] = my.number[i]
>  a$'number of coconuts in bin 7'
[1] "I have 4 coconuts"
```

Data Frames

A dataframe has been referred to as "a list of variables/vectors of the same length". In the following example, a dataframe of two vectors is created, each of five elements. The first vector, v1, is composed of a sequence of the integers 1 through 5. A second vector, v2, is composed of five logical values drawn of type T and F. The dataframe is then created, composed of the vectors. The columns of the data frame can be accessed using integer subscripts or the column name and the $ symbol.

```
> v1 <- 1:5
> v2 <- c(T,T,F,F,T)
> df <- data.frame(v1,v2)
> print(df)
  v1    v2
1  1  TRUE
2  2  TRUE
3  3 FALSE
4  4 FALSE
5  5  TRUE
> df[,1]
 [1] 1 2 3 4 5
> df$v2
 [1] TRUE TRUE FALSE FALSE TRUE
```

The dataframe may be created directly. In the following code, the dataframe is created - naming each vector composing the dataframe as part of the argument list.

```
> df <- data.frame(foo=1:5,bar=c(T,T,F,F,T))
> print(df)
  foo   bar
1   1  TRUE
2   2  TRUE
3   3 FALSE
4   4 FALSE
5   5  TRUE
```

External links

- data.frame objects in R (http://www.r-statistics.com/2011/12/data-frame-objects-in-r-via-r-in-action/) (a sample chapter from the “R in Action” book)
- Aggregation and Restructuring of data.frame objects (http://www.r-bloggers.com/aggregation-and-restructuring-data-from-%E2%80%9Cr-in-action%E2%80%9D/) (a sample chapter from the “R in Action” book)
