6.0 Objects in Splus

Notes based on those of Professor Joe Schafer

6.1 Objects

Objects are things created by and/or recognized by Splus. They include vectors, matrices, lists, and data frames. Every object carries with it some attributes that identify it—what kind of object it is, it’s size, etc.

6.2 Introduction to vectors

Let’s look at a relatively simple object, a vector. A vector is a bunch of things of the same type stuck together. Sticking numbers together produces a numeric vector.

```r
> x <- 1.7 # a vector of length 1
> x <- rnorm(100) # a vector of 100 standard normal variates
> x <- c(1,2,3,4,5,6) # integers from 1 to 6
> x <- 1:6 # same thing
> x <- rep(9.1,3) # gives (9.1,9.1,9.1)
> x <- rep(c(1,2),4) # gives (1,2,1,2,1,2,1,2)
> x <- c(99,rep(1,3)) # gives (99,1,1,1)
```

Sticking logical values (T or F) together produces a logical vector.

```r
> x <- c(T,F,F,T)
> x <- rep(c(T,F),6)
```

Sticking character strings together produces a character vector. Here’s a character vector of length 1.
> x <- "Your grandmother wears army boots."

Here's a character vector of length 3.
> x <- c("hot","warm","cold")

### 6.3 Accessing individual elements of vectors

Square brackets can be used to access individual elements of vectors. Elements can be designated by their position in the linear storage order.

> x <- rnorm(100)
> x[7]  # prints the 7th element of x
> x[2:10]  # prints elements 2,3,...,10 of x
> x[4] <- 10  # sets 4th element of x to 10
> x[7:10] <- 0  # sets elements 7, 8, 9, 10 equal to 0

In an expression like the last one ("x[7:10] <- 0"), suppose the two sides of the expression have different lengths. Then Splus will fill in the left-hand side by repeating the right-hand side over and over. If the length of the left-hand side is not a multiple of the length of the right-hand side, then Splus returns a warning message.

> x <- rnorm(10)
> x[1:10] <- c(1,2)
> x
 [1] 1 2 1 2 1 2 1 2 1 2
> x[1:9] <- c(3,4)
Warning messages:
  Replacement length not a multiple of number of elements to replace in: x[1:9] <- c(3, 4)
> x
 [1] 3 4 3 4 3 4 3 4 3 2

2
We can reorder the elements of a vector like this:

```r
> x <- c("hot","warm","cold")
> x[c(2,3,1)]
[1] "warm" "cold" "hot"
```

Negative numbers in square brackets indicate elements to be omitted.

```r
> x <- 4:7
> x
[1] 4 5 6 7
> x[-2]
[1] 4 6 7
```

Elements of a vector can also be accessed by logical values (T or F). The syntax is “x[a]” where a is a logical vector of the same length as x. The result is a vector containing those elements of x in the same positions as the T’s in a.

```r
> x <- 1:4
> x[c(T,F,F,T)]
[1] 1 4
```

The operators “==” (equal to), “!=” (not equal to), “<”, “>”, “<=” (less than or equal to), and “>=” (greater than or equal to) will return T’s or F’s according to whether the specified conditions hold or not. These can be extended with “&” (and), “!” (not), “|” (or).

```r
> x <- c(1,2,3,4)
> x==2
[1] F T F F
> x>=3
[1] F F T T
> (x>=2)&(x!=3)
[1] F T F T
> (x==1)|(x==4)
[1] F T F T
```
These logicals can be a great help in recoding data.

```r
> x <- c(1,2,2,1,3,3,1,2)
> x[x==3] <- 4
> x
[1] 1 2 2 1 4 4 1 2
> x[x==2] <- NA # NA means missing value
> x
[1] 1 NA NA 1 4 4 1 NA
```

### 6.4 Attributes of vectors

The attributes of a vector can be seen by using particular functions, such as `length()` and `storage.mode()`.

```r
> length(x) # prints length of x
> storage.mode(x) # data storage mode of x
```

The storage modes include “double” (i.e. double precision real), “single” (single precision real), “character”, “logical”, and “complex”. The default storage mode for numeric data is double precision. Try looking at the storage modes for these objects:

```r
> x <- 2
> x <- as.integer(2)
> x <- 1*x
> x <- c(T,F,T,T,F)
```

A vector can also have a `names()` attribute. The `names()` of a vector (if it exists) is a vector of character strings of the same length as the vector.

```r
> x <- c(99,100,20) # grades on an exam
> names(x) # no names yet
NULL
```
> names(x) <- c("Linda","John","Silvestre") # assign names
> names(x)
[1] "Linda" "John" "Silvestre"
> x
Linda John Silvestre
   99  100  20
> x[c(3,1,2)]
Silvestre Linda John
   20   99  100

See what happens if we change the length of a vector.

> x <- c(99,100,20)
> length(x) <- 2
> x
[1] 99 100
> length(x) <- 5
[1] 99 100   NA   NA   NA

Now see what happens if we change the storage mode.

> x <- c(1.7,pi,exp(1))
> x
[1] 1.700000 3.141593 2.718282
> storage.mode(x) <- "integer"
> x
[1] 1 3 2
> storage.mode(x) <- "character"
> x
[1] "1" "3" "2"

Sometimes we need to create vectors of a specific length and storage mode whose elements will be filled in later. We can do this using the functions double(), single(), integer(), character(), and logical().

> numeric(3)
[1] 0 0 0
> single(3)
[1] 0 0 0
> integer(3)
6.5 Miscellaneous Functions

Here are a few more functions that are useful for manipulating vectors.

> rev(4:7)                # reverses the order
[1] 7 6 5 4
> sort(c(6,9,2,7,1))      # sorts the elements in ascending order
[1] 1 2 6 7 9
> sort(c("Linda","John","Silvestre"))
[1] "John"  "Linda"  "Silvestre"
> order(c("Linda","John","Silvestre"))  # returns the sort order
[1] 2 1 3
> x[order(x)]             # same as sort(x)
> x <- c(6,5,6,6,7,5)
> duplicated(x)           # T if an element has appeared previously
[1] F F T T F T
> unique(x)               # returns elements that are not duplicated
[1] 6 5 7

For character vectors, the functions substring(), paste(), and nchar() are also highly useful. Look these up using help().

6.6 Reading data from a file

The easiest way to read in data from an external text (ASCII) file is to use scan("filename"). By default, scan() interprets any amount of white space (blanks, tabs, and possibly newlines) as separating
fields. It returns a vector. Suppose the file “mydata.dat” looks like this:

```
5 4 6
4 99 2
3.2 9 0
NA 1 NA
```

The last line of the file should consist of a single carriage return.

```R
> x <- scan("mydata.dat")
> x
[1] 5.0 4.0 6.0 4.0 99.0 2.0 3.2 9.0 0.0 NA 1.0 NA
```

6.7 Matrices

A matrix is a two-dimensional array. One way to create a matrix is to first create a vector and then turn it into a matrix using the function `matrix()`. By default, Splus fills in the matrix in a columnwise fashion (column 1 first, then column 2, then column 3, etc.) To fill in the matrix row by row, use the argument “byrow=T”.

```R
> x <- 1:10
> matrix(x,nrow=2,ncol=5)
[1,] 1 3 5 7 9
[2,] 2 4 6 8 10
> matrix(x,2,5,byrow=T)
[1,] 1 2 3 4 5
[2,] 6 7 8 9 10
```

Matrices have the attributes `nrow()`, `ncol()`, `dim()`, and `length()`. The value of `dim(x)` is `c(nrow(x),ncol(x))` and the value of `length(x)` is `nrow(x)*ncol(x)`.
In addition, a matrix can have dimnames(), which is a set of names for the rows and the columns. We'll say more about that later.

Another way to form matrices is to glue together vectors of the same length using rbind() and cbind().

> rbind(1:5,rep(9,5)) # bind together as rows

```
[,1,] 1  2  3  4  5
[,2,] 9  9  9  9  9
```

> cbind(1:5,rep(9,5),rep(NA,5))

```
  [,1] [,2] [,3]
[,1,] 1 9 NA
[,2,] 2 9 NA
[,3,] 3 9 NA
[,4,] 4 9 NA
[,5,] 5 9 NA
```

If you apply the standard algebraic operators and functions (+, -, *, /, ^, log(), sqrt(), and so on) to a matrix, the operation is performed elementwise.

> x <- matrix(1:10,2,5,byrow=T)
> x

```
[1,]  1  2  3  4  5
[2,]  8  9 10 NA NA
```

8
If the components of an expression are not of the same dimensions, then Splus expands each of them until their dimensions are the same as those of the largest one, and then computes the result in an elementwise fashion.

Matrix multiplication is performed by "%*%". The function t(x) takes the transpose of x, and solve(x) computes the inverse of x.
The following details about the "%*%" operator come from help("%*%").

Any computation involving a missing value will result in a missing value.

The last extent of x must be the same size as the first extent of y.

Vectors are not oriented, therefore a vector of length n can multiply an n by n matrix on the left or right.

If neither x nor y are matrices, then the result of x %*% y will be a one by one matrix.

The expression x * y will perform elementwise multiplication, not matrix multiplication.

### 6.8 Accessing elements of matrices

We can access elements of matrices in much the same way as we accessed elements of vectors: by using square brackets. A comma within the square brackets separates the row indices from the column indices. Look at the following examples.

```r
> x
[1,]   1   12
[2,]   2   14
[3,]   3   16
[4,]   4   18
[5,]   5   20
```
If we fail to use a comma within the square brackets, as in “x[2]” or “x[3:7]” or in “x[x==4]”, then the matrix x is treated like a vector.

> x[2]
[1] 6
> x[3:7]
[1] 11 2 7 12 3
> x[x==4] <- NA
> x

If we fail to use a comma within the square brackets, as in “x[2]” or “x[3:7]” or in “x[x==4]”, then the matrix x is treated like a vector.

> x[2]
[1] 6
> x[3:7]
[1] 11 2 7 12 3
> x[x==4] <- NA
> x

If we fail to use a comma within the square brackets, as in “x[2]” or “x[3:7]” or in “x[x==4]”, then the matrix x is treated like a vector.

> x[2]
[1] 6
> x[3:7]
[1] 11 2 7 12 3
> x[x==4] <- NA
> x

If we fail to use a comma within the square brackets, as in “x[2]” or “x[3:7]” or in “x[x==4]”, then the matrix x is treated like a vector.

> x[2]
[1] 6
> x[3:7]
[1] 11 2 7 12 3
> x[x==4] <- NA
> x

If we fail to use a comma within the square brackets, as in “x[2]” or “x[3:7]” or in “x[x==4]”, then the matrix x is treated like a vector.

> x[2]
[1] 6
> x[3:7]
[1] 11 2 7 12 3
> x[x==4] <- NA
> x

If we fail to use a comma within the square brackets, as in “x[2]” or “x[3:7]” or in “x[x==4]”, then the matrix x is treated like a vector.

> x[2]
[1] 6
> x[3:7]
[1] 11 2 7 12 3
> x[x==4] <- NA
> x

If we fail to use a comma within the square brackets, as in “x[2]” or “x[3:7]” or in “x[x==4]”, then the matrix x is treated like a vector.

> x[2]
[1] 6
> x[3:7]
[1] 11 2 7 12 3
> x[x==4] <- NA
> x

If we fail to use a comma within the square brackets, as in “x[2]” or “x[3:7]” or in “x[x==4]”, then the matrix x is treated like a vector.

> x[2]
[1] 6
> x[3:7]
[1] 11 2 7 12 3
> x[x==4] <- NA
> x

If we fail to use a comma within the square brackets, as in “x[2]” or “x[3:7]” or in “x[x==4]”, then the matrix x is treated like a vector.

> x[2]
[1] 6
> x[3:7]
[1] 11 2 7 12 3
> x[x==4] <- NA
> x

If we fail to use a comma within the square brackets, as in “x[2]” or “x[3:7]” or in “x[x==4]”, then the matrix x is treated like a vector.

> x[2]
[1] 6
> x[3:7]
[1] 11 2 7 12 3
> x[x==4] <- NA
> x

If we fail to use a comma within the square brackets, as in “x[2]” or “x[3:7]” or in “x[x==4]”, then the matrix x is treated like a vector.

> x[2]
[1] 6
> x[3:7]
[1] 11 2 7 12 3
> x[x==4] <- NA
> x

If we fail to use a comma within the square brackets, as in “x[2]” or “x[3:7]” or in “x[x==4]”, then the matrix x is treated like a vector.

> x[2]
[1] 6
> x[3:7]
[1] 11 2 7 12 3
> x[x==4] <- NA
> x