Simple Programming

Programming in `S-Plus` consists of writing new *functions* that can be used for further development of the language. The basic building blocks are the function that we have seen so far plus some other useful functions that we review in the sequel.

Comparison and Logical Operators

The main comparison and logical operators are listed below: The vectorized opera-

`==`	equal to
`>`	greater than
`!=`	not equal to
`<`	less than
`>=`	greater than or equal to
`<=`	less than or equal to
`&`	vectorized AND
`&&`	control AND
`|`	vectorized OR
`||`	control OR
`!`	not

tors evaluate the corresponding expressions element by element returning a vector of `T` of `F`.

```
> x <- seq(-1,1,length=12)
> x
 [1] -1.00000000 -0.81818182 -0.63636364 -0.45454545 -0.27272727 -0.09090909
 [7]  0.09090909  0.27272727  0.45454545  0.63636364  0.81818182  1.00000000
> x < 0 | x > 0.8
 [1] T T T T T T F F F F T T
> x < 0 & x > 0.8
 [1] F F F F F F F F F F F F
```

The control operators are used in constructing conditional statements.

Operating on Subsets of Data

There are several instances that we want to carry out computations using a preselected block of data. This method is called *subscripting* and `S-Plus` has very nice capabilities for it. In the following we will examine how it can be used for vectors.

```
>x
 [1] -1.00000000 -0.81818182 -0.63636364 -0.45454545 -0.27272727 -0.09090909
 [7]  0.09090909  0.27272727  0.45454545  0.63636364  0.81818182  1.00000000
> x[3]    # extract  the third element
[1] -0.6363636
> x[c(1,2,5)]   # extract the first, second and fifth elements.
[1] -1.0000000 -0.8181818 -0.2727273
> x[-(3:10)]    # extract all the elements except those in positions 3 to 10.
[1] -1.0000000 -0.8181818  0.8181818  1.0000000
> x[ x > 0]     # extract the elements that satisfy the condition.
 [1] 0.09090909 0.27272727 0.45454545 0.63636364 0.81818182 1.00000000
> x[ x > 0 & x < 0.5]
[1] 0.09090909 0.27272727 0.45454545
```

Subscripting can be generalized to matrix computations.

```
>A <- cbind(c(1,2,-1), c(12,15,18), c(-1,-4,-9))
> A
     [,1] [,2] [,3]
[1,]    1   12   -1
[2,]    2   15   -4
[3,]   -1   18   -9
> A[1,1]  #extracts the (1,1) element
[1] 1
> A[1,3]  # extracts the (1,3) element
[1] -1
> A[1:2,3]  #extracts the elements (1,3), (2,3)
[1] -1 -4
> A[1:2,2:3]   #extracts a two by two matrix
     [,1] [,2]
[1,]   12   -1
[2,]   15   -4
> A[,2:3]      # omission of a dimension gives the corresponding columns
     [,1] [,2]
[1,]   12   -1
[2,]   15   -4
[3,]   18   -9
> A[-1,2:3]   # use of negative indices
     [,1] [,2]
[1,]   15   -4
[2,]   18   -9
```

The discussion generalizes to lists

```
> mylist <- list(x,A)
> mylist
[[1]]:
 [1] -1.00000000 -0.81818182 -0.63636364 -0.45454545 -0.27272727 -0.09090909
 [7]  0.09090909  0.27272727  0.45454545  0.63636364  0.81818182  1.00000000

[[2]]:
     [,1] [,2] [,3]
[1,]    1   12   -1
[2,]    2   15   -4
[3,]   -1   18   -9

> mylist[[1]]
 [1] -1.00000000 -0.81818182 -0.63636364 -0.45454545 -0.27272727 -0.09090909
 [7]  0.09090909  0.27272727  0.45454545  0.63636364  0.81818182  1.00000000
> mylist[[2]]
     [,1] [,2] [,3]
[1,]    1   12   -1
[2,]    2   15   -4
[3,]   -1   18   -9
```

and data frames by using `[[]]` and $ respctively.

```
> is.data.frame(kyphosis)
[1] T
> names(kyphosis)
[1] "Kyphosis" "Age"      "Number"   "Start"
> kyphosis$Age
 [1]  71 158 128   2   1   1  61  37 113  59  82 148  18   1 168   1  78 175  80  27
[21]  22 105  96 131  15   9   8 100   4 151  31 125 130 112 140  93   1  52  20  91
[41]  73  35 143  61  97 139 136 131 121 177  68   9 139   2 140  72   2 120  51 102
[61] 130 114  81 118 118  17 195 159  18  15 158 127  87 206  11 178 157  26 120  42
[81]  36
```

Writing Function

To understand the concept of writing a new function in S-Plus, consider the following example which gives the standard deviation of a vector x:

```
>standard.deviation <- function(x)
{
sqrt(var(x))
}
> x <- rnorm(100, mean=0, sd=2) #100 observations from normal
```

```
                                #with mean 0 and variance 4
> var(x)
[1] 3.879332
> standard.deviation(x)
[1] 1.969602
```

Hence, to obtain the `standard.deviation` notice that we used two existing functions, the `sqrt` and `var`. This is a fundamental idea when there is need for defining a new function in `S-Plus`.

Here are some basic constructions:

`if` (A) B	evaluates A, if T evaluates B
`if` (A) B1 `else` B2	evaluates A, if T evaluates B1, else evaluates B2
`ifelse`(A,B1,B2)	vectorized `if` statement
`break`	terminates current loop
`next`	terminates current loop and starts next iteration
`return`(A)	terminates current function and returns A
`while` (A) B	evaluates A; if true evaluates B repeatedly
`repeat` A	simpler version of `while`
`for` (index `in` A) B	loop but it is computationally expensive

Some additional constructions are `switch()` and `stop`. The following example illustrate the concepts.

The first example illustrates how we can use the `if` function to generate samples from different distributions.

```
random.gener <- function(n, distribution, shape)
{
# a function to generate n random numbers
if(distribution=="gamma") rgamma(n, shape) else
if(distribution=="exp")   rexp(n) else
if(distribution=="norm") rnorm(n) else
stop("Invalid Distribution")
}
> random.gener(10, "gamma", 2)
 [1] 0.3461286 2.0791867 3.2288429 4.3973702 1.7676279 2.7317868 0.4084932 2.4203665
 [9] 0.7430161 5.1688287
> random.gener(10, "unif", 2)
Error in random.gener(10, "unif", 2): Invalid Distribution
```

This example shows how we can use the `for` and the `if` function to get the sign of a real number.

```
new.sign <-  function(x)
{
   for (i in 1:length(x)){
         if(x[i] > 0)
            x[i] <- 1
    else if(x[i] < 0)
```

```
            x[i] <- -1
   }
   x
}
> new.sign(-10:5)
 [1] -1 -1 -1 -1 -1 -1 -1 -1 -1 -1  0  1  1  1  1  1
```

However a better way is the following which avoids iterations.

```
sgnfunction <- function(x)
{
ifelse(x > 0, 1, ifelse(x<0, -1, 0))
}
> sgnfunction(-10:10)
[1] -1 -1 -1 -1 -1 -1 -1 -1 -1 -1  0  1  1  1  1  1  1  1  1  1  1
```