Appendix
An Aide Memoir for R and S-PLUS®

1. Elementary Commands

Elementary commands consist of either expressions or assignments. For example, typing the expression

```
> 42 + 8
```

in the **Commands** window and pressing **Return** will produce the following output:

```
[1]   50
```

In the remainder of this chapter, we will show the command (preceded by the prompt >) and the output as they would appear in the **Commands** window together like this:

```
> 42 + 8
[1]   50
```

Instead of just evaluating an expression, we can assign the value to a scalar using the syntax `scalar <- expression`

```
> x <- 42 + 8
```

Longer commands can be split over several lines by pressing Return before the command is complete. To indicate waiting for completion of a command, a "+" occurs instead of the > prompt. For illustration, we break the line in the assignment above:

```
> x<-
+ 48+8
```

2. Vectors

S-PLUS®

A commonly used type of R and S-PLUS® object is a *vector*. Vectors may be created in several ways of which the most common is via the concentrate command, `c`, which combines all values given as arguments to the function into a vector. For example,

```
>x<-c(1, 2, 3,4)
>x
[1] 1 2 3 4
```

Here, the first command creates a vector and the second command, *x*, a short-form for `print(x)`, causes the contents of the vector to be printed. (Note that R and S-PLUS are case sensitive, and so, for example, x and X are different objects.) The number of elements of a vector can be determined using the `length()` function:

: assignments. For example,

```
>length(x)
[1] 4
```

The `c` function can also be used to combine *strings* which are denoted by enclosing them in "." For example,

```
>names <-c ("Brian", "Sophia", "Harry")
>names
[1] "Brian" "Sophia" "Harry"
```

will produce the following

The `c()` function also works with a mixture of numeric and string values, but in this case, all elements in the resulting vector will be converted to strings as in the following.

and (preceded by the prompt
ıands window together like

```
> mix <-c(names, 55, 33)
> mix
[1] "Brian" "Sophia"    "Harry" "55"    "33"
```

Vectors consisting of regular sequences of numbers can be created using the `seq()` function. The general syntax of this function is `seq(lower, upper, increment)`. Some examples are given below:

n the value to a scalar using

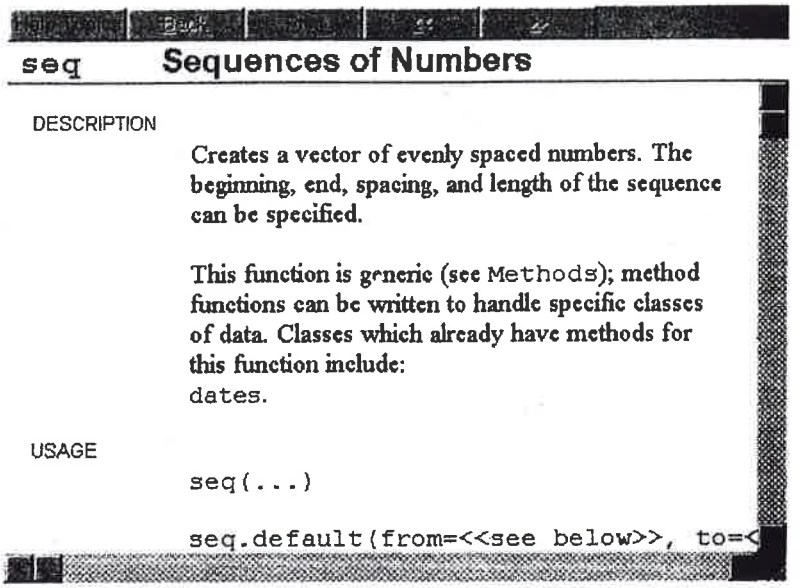
```
>seq (1, 5, 1)
[1] 1 2 3 4 5
>seq (2, 20, 2)
[1] 2 4 6 8 10 12 14 16 18 20
>x <-c(seq(1, 5, 1), seq (4, 20, 4))
>x
[1] 1 2 3 4 5 4 8 12 16 20
```

pressing Return before the
etion of a command, a "+"
ak the line in the assignment

When the increment argument is one it can be left out of the command. The same applies to the lower value. More information about the `seq` function and all other R and S-PLUS functions can be found using the `help` facilities, e.g.,

```
>help(seq)
```

shows the following information:



Sequences with increments of one can also be obtained using the syntax `first:last`, for example,

```
>1:5
[1] 1 2 3 4 5
```

A further useful function for creating vectors with regular patterns is the `rep` function, with general form `rep(pattern, number of times)`. For example,

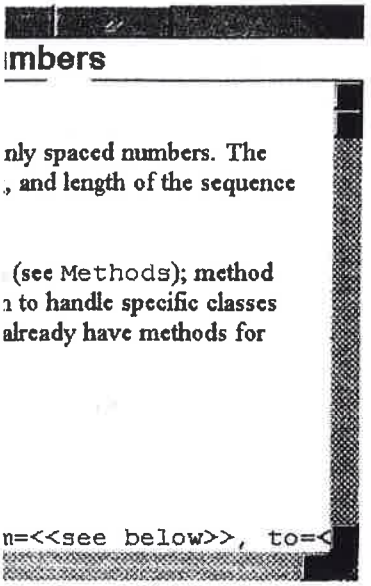
```
>rep(10, 5)
[1] 10 10 10 10 10
>rep (1:3, 3)
[1] 1 2 3 1 2 3 1 2 3
> x <- rep(seq(5), 2)
> x
[1] 1 2 3 4 5 1 2 3 4 5
```

The second argument of `rep` can also be a vector of the same length as the first argument to indicate how often each element of the first argument is to be repeated as shown in the following;

```
> x <- rep(seq (3), c(1, 2, 3))
> x
[1] 1 2 2 3 3 3
```

Increasingly complex vectors can be built by repeated use of the `rep` function

```
> x <- rep (seq (3), rep (3, 3))
> x
[1] 1 1 1 2 2 2 3 3 3
```



) be obtained using the syntax `first:`

with regular patterns is the `rep` func-
nber of times)`. For example,

vector of the same length as the first
of the first argument is to be repeated

by repeated use of the `rep` function

We can access a particular element of a vector by giving the required position in square brackets- here are two examples

```
> x <- 1:5
> x[3]
[1] 3

>x[c(1, 4)]
[1] 1 4
```

A vector containing several elements of another vector can be obtained by giving a vector of required positions in square brackets:

```
> x[c(1, 3)]
[1] 1 3
> x[1:3]
[1] 1 2 3
```

We can carry out any of the arithmetic operations described in Table A.1 between two scalars, a vector and a scalar or two vectors. An arithmetic operation between two vectors returns a vector whose elements are the results of applying the operation to the corresponding elements of the original vectors. Some examples follow:

```
> x<- 1:3
> x+2
[1] 3 4 5

>x + x
[1] 2 4 6
> x* x
[1] 1 4 9
```

We can also apply mathematical functions such as the square root or logarithm, or the others listed in Table A.2, to vectors. The functions are simply applied to each element of the vector. For example,

```
> x <- 1:3
> sqrt (x*x)
[1] 1 2 3
```

Table A.1 Arithmetic Operators

Operator	Meaning	Expression	Result
+	Plus	2 + 3	5
−	Minus	5 − 2	3
*	Times	5 * 2	10
/	Divided by	10/2	5
^	Power	2 ^ 3	8

Table A.2 Common Functions

S-PLUS function	Meaning
`sqrt ()`	Square root
`log ()`	Natural logarithm
`log10 ()`	Logarithm base 10
`exp ()`	Exponential
`abs ()`	Absolute value
`round ()`	Round to nearest integer
`ceiling ()`	Round up
`floor ()`	Round down
`sin (), cos (), tan ()`	sine, cosine, tangent
`asin (), acos (), atan ()`	arc sine, arc cosine, arc tangent

3. Matrices

Matrix objects are frequently needed in R and S-PLUS and can be created by the use of the `matrix` function. The general syntax is

```
matrix (data, nrow, ncol, byrow = F)
```

The last argument specifies whether the matrix is to be filled row by row or column by column and takes on a logical value. The expression *byrow=F* indicates that `F` (false) is the default value. An example follows;

```
> x <-c(1, 2, 3)
> y <-c(4, 5, 6)
> xy <- matrix (c(x, y), nrow =2)
> xy
     [,1]  [,2]  [,3]
[1,]    1     3     5
[2,]    2     4     6
```

Here the number of columns is not specified and so is determined by simple division:

```
> xy <-matrix (c(x, y), nrow = 2, byrow =T)
xy

     [,1]  [,2]  [,3]
[1,]    1     2     3
[2,]    4     5     6
```

Here the matrix, is filled row-wise instead of by columns by setting the `byrow` argument to `T` for True. A square bracket with two numbers separated by a comma is used to refer to an element of a matrix. The first number specifies the row, and the second specifies the column.

```
>xy [1, 3]
[1] 3
```

The `[i,]` and `[,j]` nomenclature is used to refer to complete rows or columns of a matrix and can be used to extract particular rows or columns as shown in the following examples;

Meaning
Squ ot
Nat garithm
Logarithm base 10
Exponential
Absolute value
Round to nearest integer
Round up
Round down
sine, cosine, tangent
arc sine, arc cosine, arc tangent

```
> xy [1,]
[1] 1 2 3
> xy [,2]
[1] 2 5

>xy [, c(1, 3)]

     [,1] [,2]
[1,]    1    3
[2,]    4    6
```

As with vectors, arithmetic operations operate element by element when applied to matrices, for example;

```
> xy* xy

     [,1] [,2] [,3]
[1,]    1    4    9
[2,]   16   25   36
```

Matrix multiplication is performed using the `%*%` operation as here

```
> xy %*% t(xy)

     [,1] [,2]
[1,]   14   32
[2,]   32   77
```

Here the matrix *xy* is multiplied by its transpose (obtained using the *t*() function). An attempt to apply matrix multiplication to *xy* by *xy* would, of course, result in an error message. It is usually extremely helpful to attach names to the rows and columns to a matrix. This can be done using the `dimension()` function. We shall illustrate this in Section 5 after we have covered `list` objects.

As with vectors, matrices can be formed from numeric and string objects, but in the resulting matrix, all elements will be strings as illustrated below:

```
> Mix <- matrix(c(names, 55, 32, 30), nrow = 2
+ byrow = T)
> Mix

     [,1]      [,2]      [,3]
[1,] "Brian"   "Sophia"  "Harry"
[2,] "55"      "32"      "30"
```

Higher dimensional matrices with up to eight dimensions can be defined using the `array()` function.

4. Logical Expressions

So far, we have mentioned values of type numeric or character (string). When a numeric value is missing, it is of type NA. (Complex numbers are also available.) Another type in R and S-PLUS is logical. There are two logical values, `T` (true) and

ıd S-PLUS and can be created by the
:ax is

F)

x is to be filled row by row or column
expression *byrow=F* indicates that F
ws;

d sc ctermined by simple division:

byrow =T)

ıf by columns by setting the byrow
ı two numbers separated by a comma
e first number specifies the row, and

› refer to complete rows or columns
lar rows or columns as shown in the

Table A.3 Logical Operators

Operator	Meaning
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to
==	Equal to
!=	Not equal to
&	And
|	Or
!	not

F (false), and a number of logical operations that are extremely useful when making comparisons and choosing particular elements from vectors and matrices.

The symbols used for the logical operations are listed in Table A.3. We can use a logical expression to assign a logical value (T or F) to x:

```
> x <-3 = = 4
> x
[1] F
> x <-3 < 4
> x
[1] T
> x < - 3 = = 4 & 3 < 4
> x
[1] F
> x < - 3 = = 4 | 3 < 4
> x
[1] T
```

In addition to logical operators, there are also logical functions. Some examples are given below:

```
> is.numeric (3)
[1] T
> is.character (3)
[1] F
> is.character ("3")
[1] T
> 1/0
[1] Inf
>is.numeric (1/0)
[1] T
>is.infinite (1/0)
[1] T
```

Logical operators or functions operate on elements of vectors and matrices in the same say as arithmetic operators:

```
> is.na(c(1, 0, NA, 1))
[1] F F T F
```

Meaning
s than
eater than
Less than or equal to
Greater than or equal to
Equal to
Not equal to
And
Or
not

that are extremely useful when making
its from vectors and matrices.
ns are listed in Table A.3. We can use
(T or F) to x:

logical functions. Some examples are

ements of vectors and matrices in the

```
> ! is.na (c(1, 0, NA, 1))
[1] T T F T
> x <- seq(20)
> x <10
[1] T T T T T T T T T F F F F F F F F F F
```

A logical vector can be used to extract a subset of elements from another vector as follows:

```
> x[x <10]
[1] 1 2 3 4 5 6 7 8 9
```

Here, the elements of the vector less than 10 are selected as the values corresponding to T in the vector $x < 10$. We can also select element in x depending on the values in another vector y:

```
> x <-seq(50)
> y <- c(rep(0, 10), rep(1, 40))
> x[y = =0]
[1]  1 2 3 4 5 6 7 8  9 10
```

5. List Objects

`List` objects allow any other R or S-PLUS objects to be linked together. For example,

```
>x<-seq(10)
>y<- matrix(seq(10), nrow = 5
>xylist<-list (x,y)
>xylist
[[1]]:
[1] 1 2 3 4 5 6 7 8 9 10

[[2]]:

     [,1]  [,2]
[1,]    1     6
[2,]    2     7
[3,]    3     8
[4,]    4     9
[5,]    5     10
```

Note the elements of the list are referred to by a double square brackets notation; so we can print the first component of the list using

```
>xylist[[1]]
[1] 1 2 3 4 5 6 7 8 9 10
```

The components of the list can also be given names and later referred using the `list$name` notation,

```
>xylist <-list (X=x, Y=y)
>xylist$X
[1] 1 2 3 4 5 6 7 8 9 10
```

List objects can, of course, include other list objects

```
>newlist<-list(xy=xylist, z=rep(0,10))
>newlist$xy
$X
[1] 1 2 3 4 5 6 7 8 9 10

$Y:
     [,1] [,2]
[1,]    1    6
[2,]    2    7
[3,]    3    8
[4,]    4    9
[5,]    5   10

>newlist$z
[1] 0 0 0 0 0 0 0 0 0 0
```

The rows and columns of a matrix can be named using the `dimnames()` function and a list object

```
>x<-matrix(seq(12), nrow=4)
> dimnames(x)<-list(c("R1","R","R3","R4"),
+c("C1", "C2", "C3"))
>x

   C1 C2 C3
R1  1  5  9
R2  2  6 10
R3  3  7 11
R4  4  8 12
```

The names can be created more efficiently by using the `paste()` function, which combines different strings and numbers into a single string

```
> dimnames(x)<-list(paste("row", seq (4)),
+paste ("col", seq(3)))
>x
     col 1 col 2 col 3
row1     1     5     9
row2     2     6    10
row3     3     7    11
row4     4     8    12
```

Having named the rows and columns, we can, if required, refer to elements of the matrix using these names,

```
>x["row 1", "col 3"]
[1] 9
```

6. Data Frames

Data sets in R and S-PLUS are usually stored as matrices, which we have already met, or as data frames, which we shall describe here.

Data frames can bind vectors of different types together (e.g., numeric and character), retaining the correct type of each vector. In other respects, a data frame is like a matrix so that each vector should have the same number of elements. The syntax for creating a data frame is `data.frame(vector1,vector 2, ...)`, and an example of how a small data frame can be created is as follows:

```
>height<-c(50, 70, 45, 80, 100)
>weight<-c(120, 140, 100, 200, 190)
>age<-c(20, 40, 41, 31, 33)
>names<-c("Bob", "Ted", "Alice", "Mary", "Sue")
sex<-c("Male", "Male", "Female", "Female")
>data<-data.frame(names, sex, height, weight, age)
>data
```

	names	sex	height	weight	age
1	Bob	Male	50	120	20
2	Ted	Male	70	140	40
3	Alice	Female	45	100	41
4	Mary	Female	80	200	31
5	Sue	Female	100	190	33

Particular parts of a data frame can be extracted in the same way as for matrices

```
>data[,c(1,2,5)]
```

	names	sex	age
1	Bob	Male	20
2	Ted	Male	40
3	Ali	Female	41
4	Mary	Female	31
5	Sue	Female	33

Column names can also be used

```
>data[,"age"]
[1] 20 40 41 31 33
```

Variables can also be accessed as in lists:

```
>data$age
[1] 20 40 41 31 33
```

It is, however, more convenient to "attach" a data frame and work with the column names directly, for example,

```
>attach (data)
>age
[1] 20 40 41 31 33
```

Note that the `attach()` command places the data frame in the 2nd position in the search path. If we assign a value to `age`, for example,

```
>age <-10
>age
[1] 10
```

This creates a new object in the first position of the search path that "masks" the `age` variable of the data frame. Variables can be removed from the first position in the search path using the `rm()` function:

```
>rm(age)
```

To change the value of age within the data frame, use the syntax

```
>data$age<-c(20, 30, 45, 32, 32)
```