CHAPTER 11

Descriptive Functions and Manipulating Objects

For arrays, matrices, vectors, lists, and expressions, in command line R, there are a number of functions that describe various attributes of an object. In R Studio, many attributes, such as the number of columns in a matrix or the length of a list, are given to the right of the object name under the "Environment" tab in the upper right window.

Also, there are a number of functions that manipulate objects to create new objects. The functions covered in this chapter are the descriptive functions `dim()`, `nrow()`, `NROW()`, `ncol()`, `NCOL()`, `length()`, `nchar()`, and `nzchar()`; the functions that manipulate objects: `cbind()` and `rbind()`; the apply functions, `sweep()`, `scale()`, and `aggregate()`; the table functions and the functions `tabulate()`, and `ftable()`; and the string functions: `grep()`, `grepl()`, `agrep()`, `grepRaw()`, `sub()`, `gsub()`, `regexpr()`, `gregexp()`, `regexec()`, `substr()`, `substring()`, and `strsplit()`.

Descriptive Functions

The descriptive functions describe qualities of objects. This section discusses some descriptive functions that are useful when writing functions or creating objects. The functions are `dim()`, `nrow()`, `ncol()`, `NROW()`, `NCOL()`, `length()`, and `nchar()`.

The Function dim()

For objects for which dimensions make sense—such as matrices, data. frames, tables, or arrays—the function `dim()` returns the number of levels in each of the dimensions of the object. For objects of other classes, `dim()` returns **NULL**. An example follows:

```
> a = 1:2
> b = 1:3

> dim( a )
NULL

> a %o% b %o% a
, , 1

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

, , 2

     [,1] [,2] [,3]
[1,]    2    4    6
[2,]    4    8   12

> dim( a %o% b %o% a )
[1] 2 3 2
```

The dimensions of the object can be changed if the product of the original dimensions equals the product of the dimensions of the result. An example follows:

```
> a.ar = a %o% b

> a.ar
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    2    4    6

> dim( a.ar )
[1] 2 3

> dim( a.ar )= c( 3, 2 )

> a.ar
     [,1] [,2]
[1,]    1    4
[2,]    2    3
[3,]    2    6
```

You can find more information about `dim()` by entering **?dim** at the R prompt or by using the Help tab in R Studio.

The Functions nrow(), ncol(), NROW(), and NCOL()

For matrices, data.frames, and arrays, `nrow()` and `ncol()` give the number of levels in the first and second dimensions of the matrix, data frame, or array, respectively. Other classes of objects return **NULL**. An example follows, using the a and b of the last section:

```
> a %o% b
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    2    4    6

> nrow( a %o% b )
[1] 2

> ncol( a %o% b )
[1] 3

> nrow( 1:20 )
NULL
```

Sometimes vectors must be treated as matrices or arrays. The functions `NROW()` and `NCOL()` treat vectors as one-column matrices but otherwise are the same as `nrow()` and `ncol()`. An example follows:

```
> NROW( 1:20  )
[1] 20

> NCOL( 1:20 )
[1] 1
```

You can find more information about `nrow()`, `ncol()`, `NROW()`, and `NCOL()`by entering **?nrow** at the R prompt or by using the Help tab in R Studio.

The Function length()

The next descriptive function we will explain is `length()`. The argument to `length()` can be any mode of object. For atomic objects, `length()` returns the number of elements in the object. For list objects, `length()` returns the number of the lowest level elements. For functions, `length()` returns one. For calls, `length()` returns the number of arguments entered in the creation of the call. For expressions, `length()` returns the number of elements in the expression. Some examples follow:

```
> a.mat=matrix( 1:4, 2, 2 )

> a.mat
     [,1] [,2]
[1,]    1    3
[2,]    2    4

> length( a.mat )
[1] 4

> a.list=list( mat, c( "abc", "cde" ) )

> a.list
[[1]]
     [,1] [,2]
[1,]    1    3
[2,]    2    4

[[2]]
[1] "abc" "cde"

> length( a.list )
[1] 2

> a.fun = function( mu, se=1, alpha=.05 ){
  z_value = qnorm( 1-alpha/2, mu, se )
  print( z_value )
}
> length( a.fun )
[1] 1

> a.call=call( "lm", y~x )

> a.call
lm(y ~ x)
```

```
> length( a.call )
[1] 2
1

> a.exp = expression( a.call, sin( 1:5/180 * pi ) )

> a.exp
expression(a.call, sin(1:5/180 * pi))

> length(a.exp)
[1] 2
```

The length of an atomic or list object can be assigned using `length()`. For other mode objects, an attempted `length()` assignment returns an error. If **n** is the length of an atomic object, then setting the length to a value larger than **n** generates **NA**s for the extra elements. Setting the length shorter than **n** removes elements. In either case, a vector is returned unless the length is not changed, in which case the original object is returned. An example follows:

```
> a.mat
     [,1] [,2]
[1,]    1    3
[2,]    2    4

> a.mat.2 = a.mat

> length( a.mat.2 )=6

> a.mat.2
[1]  1  2  3  4 NA NA

> a.mat.2 = a.mat

> length( a.mat.2 )=3

> a.mat.2
[1] 1 2 3
```

```
> a.mat.2 = a.mat

> length( a.mat.2 )=4

> a.mat.2
     [,1] [,2]
[1,]    1    3
[2,]    2    4
```

For objects of mode `list`, lengthening the list adds **NULL** elements at the lowest level while shortening the list removes elements at the lowest level. An example follows:

```
> a.list
[[1]]
     cl1 cl2
[1,]   1   3
[2,]   2   4

[[2]]
[1] "abc" "cde"

> length( a.list )=3

> a.list
[[1]]
     cl1 cl2
[1,]   1   3
[2,]   2   4

[[2]]
[1] "abc" "cde"

[[3]]
NULL
```

```
> length( a.list )=1

> a.list
[[1]]
     cl1 cl2
[1,]   1   3
[2,]   2   4
```

You can find more information about `length()` by entering **?length** at the R prompt or by using the Help tab in R Studio.

The Functions nchar() and nzchar()

The function `nchar()` counts characters in objects that can be coerced to mode `character`. The function nzchar() returns a logical vector indicating which elements contain non-empty strings.

The function nchar() takes four arguments: **x**, **type**, **allowNA**, and **keepNA**. The argument **x** is the object. The function coerces the object to character, and the characters to be counted are the characters in each element of the coerced object. For example, redefining a.list as defined in the last section:

```
> a.list = list( matrix( 1:4, 2,2 ), c( "abc", "cde" ), NULL )

> a.list
[[1]]
     [,1] [,2]
[1,]    1    3
[2,]    2    4

[[2]]
[1] "abc" "cde"

[[3]]
NULL
```

```
> as.character( a.list )
[1] "1:4" "c(\"abc\", \"cde\")" "NULL"
```

```
> nchar( a.list )
[1]  3 15  4
```

Quotes are not counted.

The argument **type** is a character argument and can take on the values of **"bytes," "chars,"** or **"width."** If **"bytes"** is chosen, the bytes of the strings are counted. If **"chars"** is chosen, the standard text number of characters is counted. If **"width"** is chosen, the number of characters that the function `cat()` would assign the strings is counted. The default value is **"chars."** Usually, there is no difference between the three.

The argument **allowNA** is a logical argument. If set equal to **TRUE**, strings that are not valid are set equal to **NA**. If set equal to **FALSE**, strings that are not valid give an error and cause the function to stop. The default value is **FALSE**.

The argument **keepNA** is a logical argument that tells nchar() whether to convert NAs to character strings or to keep them as **NA**s. The default value is **NA**, which tells nchar() to set the argument to TRUE if **type** is "bytes" or "char" and to FALSE if **type** is "width". If the argument is a data frame, since a data frame is a list, each column is converted to a character string, and the NAs are also made into character strings, before the counting done by nchar(), whether keepNA is set to TRUE or FALSE. For vectors, matrices, and arrays, NAs are not converted to strings.

For example:

```
> a.df=data.frame( 1, NA, 12 )
```

```
> as.character( a.df )
[1] "1"  "NA" "12"
```

```
> nchar( a.df, keepNA=F )
 X1 NA. X12
  1   2   2
```

```
>  nchar( a.df, keepNA=T )
 X1 NA. X12
  1   2   2

> a.mat = as.matrix( a.df )

> as.character( a.mat )
[1] "1"  NA   "12"

> nchar( a.mat, keepNA=F )
     X1 NA. X12
[1,]  1   2   2

> nchar( a.mat, keepNA=T )
     X1 NA. X12
[1,]  1  NA   2
```

The function nzchar() gives a logical vector of the same length as the object assigned to the first argument. The function returns a vector of TRUEs, FALSEs, and NAs that depend on whether an element is a nonempty string, and empty string or is missing. The function takes two arguments, **x** and **keepNA**. The argument **x** is an object that can be coerced to a character vector. The argument **keepNA** is logical and can take on the values TRUE, FALSE or NA. If **keepNA** is TRUE, NA's return NA's, if FALSE or NA, NAs returns TRUE. The default value is FALSE.

For example:

```
> nzchar( c( "1", NA, "12", "" ), keepNA=F )
[1]  TRUE  TRUE  TRUE FALSE

> nzchar( c( "1",  NA,   "12", "" ), keepNA=NA )
[1]  TRUE  TRUE  TRUE FALSE

> nzchar( c( "1",  NA,   "12", "" ), keepNA=T )
[1]  TRUE    NA  TRUE FALSE
```

You can find more information about `nchar() and nzchar()` by entering **?nchar** at the R prompt or by using the Help tab in R Studio.

Manipulating Objects

There are a number of functions that manipulate R objects and make programming easier. This subsection covers some of the functions, including `cbind()`, `rbind()`, `apply()`, `lapply()`, `sapply()`, `vapply()`, `tapply()`, `mapply()`, eapply(), `sweep()`, `scale()`, `aggregate()`, `table()`, `tabulate()`, and `ftable()`.

The Functions cbind() and rbind()

The functions `cbind()` and `rbind()` are self-explanatory for vectors, matrices, and data frames. The function `cbind()` binds columns. The function `rbind()` binds rows.

For lists that are not matrixlike, the functions return the type and number of elements in each of the lowest level elements of the list, creating a matrix of the types. Lists can be bound with nonlist objects. The result will be a list, but the nonlist arguments will not be converted like the list part of the result.

In the call to the function, the objects to be bound are separated by commas. For `cbind()`, vectors are treated as columns. For `rbind()`, vectors are treated as rows.

For vectors, vectors being bound do not have to be of the same length. The vectors cycle with themselves and with higher dimensional objects. For higher dimensional objects, the objects will not cycle. If, for `rbind()`, the numbers of columns do no match or, `for cbind()`, the numbers of rows do not match, an error is given.

The resulting object takes on the type of the highest level object entered, where the hierarchy, from lowest to highest, is `raw, logical, integer, double, complex, character,` and `list`.

There is one argument to `cbind()` and `rbind()` other than the objects to be bound—the argument **deparse.level**, which is used to create labels for objects that are not matrixlike. The argument is an integer argument and can take on the values of **0**, **1**, or **2**, although any value that can be coerced to an integer works. Values that do not give **1** or **2** when coerced to an integer give the same result as **0**. The default value is **1**.

For data frames, if a data frame is included in the objects to be bound and a list that is not a data frame is not included, then the result is a data frame. In that case, any character columns are changed to factors unless specified to not.

For time series, `cbind()` gives a multivariate time series, whereas for `rbind()`, the time series reverts to a plain matrix.

An example follows:

```
> ab.list = list( one=1:3, two=1:5 )

> ab.list
$one
[1] 1 2 3

$two
[1] 1 2 3 4 5

> cbind( ab.list, 1:2 )
    ab.list
one Integer,3 1
two Integer,5 2

> cbind( ab.list, 1:2, deparse.level=0 )
    [,1]      [,2]
one Integer,3 1
two Integer,5 2
```

```
> cbind( ab.list, 1:2, deparse.level=2 )
    ab.list   1:2
one Integer,3 1
two Integer,5 2
```

You can find more information about cbind() and rbind() by entering **?cbind** at the R prompt or by using the Help tab in R Studio.

The Apply Functions

There are several functions in R for applying a function over a subset of an object, seven of which are covered here. The seven functions are `apply()`, `lapply()`, `sapply()`, `vapply()`, `tapply()`, mapply(), and `eapply()`. The functions to be applied can be user defined, which can be quite useful.

The Function apply()

The function `apply()` takes three arguments—**X**, **MARGIN**, and **FUN**—as well as any arguments to the function `FUN`. The first argument, **X**, is an array (including matrices). The second argument gives the margin(s) over which the function is to operate, and `FUN` is the function to be applied.

For matrices, entering **1** for **MARGIN** applies the function across the columns. For **2**, the function is applied down the rows.

The function to be applied is entered without parentheses. Any arguments to the function are entered next, separated by commas. The result is an array, matrix, or vector. An example follows:

```
> a.mat=matrix( 1:4, 2, 2, dimnames=list( c( "r1", "r2" ),
                                          c( "c1", "c2" ) ) )
> a.mat
   c1 c2
r1  1  3
r2  2  4
```

```
> apply( a.mat, 1, sum )
r1 r2
 4  6

> apply( a.mat, 1, pnorm, 3, 1 )
           r1        r2
c1 0.02275013 0.1586553
c2 0.50000000 0.8413447
```

In the example, the first apply finds the sums of the rows. For the second apply, the arguments to `pnorm()` are the rows in mat for the **q** values, **3** for the value of **mean**, and **1** for the value of **sd**. Note that the matrix is transposed in the result.

You can find more information about `apply()` by entering **?apply** at the R prompt or by using the Help tab in R Studio.

The lapply(), sapply(), and vapply() Functions

The `lapply()`, `sapply()`, and `vapply()` functions work with vectors, including lists, and expressions. If **X** is not a list, then **X** is coerced to a list. The elements must be of the correct mode for the function being applied.

The function `lapply()` is the simplest with just two arguments plus any arguments to the function to be applied. The function `sapply()` takes four arguments plus any extra arguments for the function to be applied. The function `vapply()` also takes four arguments plus any extra for the function to be applied.

The Function lapply()

The function `lapply()` takes the arguments **X** and **FUN**, plus any extra arguments for **FUN**. The function `FUN` is applied to every element of the vector or to every second level element of the list. The result is a list.
An example follows:

```
> b.list=list( 1:7, 3:4 )

> b.list
[[1]]
[1] 1 2 3 4 5 6 7

[[2]]
[1] 3 4

> lapply( b.list, sum )
[[1]]
[1] 28

[[2]]
[1] 7
```

You can enter arithmetic operators by enclosing the operators within quotes. For example:

```
> lapply( 1:2, "^", 2 )
[[1]]
[1] 1

[[2]]
[1] 4
```

The Function sapply()

The function `sapply()` also operates on vectors, including lists, and expressions. The function takes the arguments **X** and **FUN**, then any arguments to **FUN** followed by the arguments **simplify** and **USE.NAMES**.

The argument **simplify** can be logical or the character string **"array"**. The argument **simplify** tells `sapply()` to simplify the list to a vector or matrix if **TRUE**, and to an array if set equal to **"array"**. No simplification is done if set equal to **FALSE**. For **FALSE**, a list is returned. The value **TRUE** is the default.

The argument **USE.NAMES** is a logical argument. For an object of mode `character`, the argument **USE.NAMES** tells `sapply()` to use the elements of the object as names for the result. The default value is **TRUE**. An example follows:

```
> ac.list = list( one=1:5, two=3:7 )

> ac.list
$one
[1] 1 2 3 4 5

$two
[1] 3 4 5 6 7

> sapply( ab.list, sum )
one two
 15  25

> a.char = paste0( "a", 7:10 )

> a.char
[1] "a7"  "a8"  "a9"  "a10"

> sapply( a.char, paste, "b", sep="" )
    a7     a8     a9    a10
 "a7b"  "a8b"  "a9b" "a10b"

> sapply( a.char, paste,  "b", sep="", USE.NAMES=F )
[1] "a7b"  "a8b"  "a9b"  "a10b"
```

The Function vapply()

The function `vapply()` takes the arguments **X**, **FUN**, **FUN.VALUE**, any arguments to **FUN**, and **USE.NAMES**, in that order.

The argument **FUN.VALUE** is a structure for the output from the function. The structure is the structure of the result of applying **FUN** to a single element of **X**. Dummy values of the correct mode are used in the structure. The number and mode of the dummy elements must be correct. Any extra arguments for **FUN** are placed after **FUN.VALUE**. The default value of **USE.NAMES** is **TRUE**. An example follows:

```
> set.seed( 382765 )

> ab.val=1:2

> vapply( ab.val, rnorm, matrix( .1, 2, 2 ), n=4, sd=1 )
, , 1

         [,1]      [,2]
[1,] 1.701435 1.1422971
[2,] 2.068151 0.9604146

, , 2

          [,1]     [,2]
[1,] 0.3541925 1.186276
[2,] 2.6841000 1.745577
```

In the example, **ab.val** is a vector of means entered into the function `rnorm()`, and the other arguments to `rnorm()` are **n=4** and **sd=1**.

The function `vapply()` returns an array, matrix, or vector of objects of the kind given by the argument **FUN.VALUE**.

You can find more information about `lapply()`, `sapply()`, and `vapply()` by entering **?lapply** at the R prompt or use the Help tab in R Studio.

The Function tapply()

The function `tapply()` applies functions to cross-tabulated data. The arguments are **X**, **IND**, **FUN**, any extra arguments to **FUN**, **default**, and **simplify**. The default value for **FUN** is **NULL**, the default value for **default** is NA, and the default value of **simplify** is **TRUE**.

The argument **X** must be an atomic object and is coerced to a vector. The argument can be a contingency table created by `table()`. The length of **X** is then the product of the dimensions of the contingency table.

The argument **IND** must be a vector that can be coerced to a factor or a list of vectors that can be coerced to factors. The length of **X** and the length(s) of the factor vectors must all be the same.

The values of **X** are the number of observations with a given factor combination, where the factor combinations are given by juxtaposing the factor values. If combinations are repeated, the function does not work right. There is no need to enter zeroes for factor combinations without observations, but zeroes may be included.

Using `tapply()` without a function gives the index of the cells that contain observations, while using a function gives the factor cross table, with the function applied to the contents of the cells. An example follows:

```
> cbind( c( "a", "b", "b", "c" ), c( 5, 5, 6, 5) )
     [,1] [,2]
[1,] "a"  "5"
[2,] "b"  "5"
[3,] "b"  "6"
[4,] "c"  "5"

> tapply( 1:4, list( c( "a", "b", "b", "c" ), c( 5, 5, 6, 5) ) )
[1] 1 2 5 3

> tapply( 1:4, list( c( "a", "b", "b", "c" ), c( 5, 5, 6, 5) ),
          "^", 3 )
   5  6
a  1 NA
b  8 27
c 64 NA
```

In this example, the four observations are in the cells a5, b5, b6, and c5, as can be seen by juxtaposing the two factor columns. There are six possible cells, a5, b5, c5, a6, b6, and c6. The first call to tapply() gives the cell identifiers for the four table counts. The second call applies the cube function to the table counts and prints out a full table of the results, returning NA for empty cells.

You can find more information about `tapply()`by entering **?tapply** at the R prompt or by using the Help tab in R Studio.

The Function mapply()

The function `mapply()` takes an object that is an atomic vector or a list as an argument and applies a function to each element of the vector or list. If an object that is not an atomic vector or list is entered, `mapply()` attempts to coerce the object to an atomic vector or list. The elements of the resulting object must be legal for the function to be applied. The result of `mapply()` is an atomic vector, matrix, or list.

The arguments to `mapply()` are **FUN**, **...**, **MoreArgs**, **SIMPLIFY**, and **USE.NAMES**. The argument **FUN** is the function to be applied. The argument **...** refers to the atomic vectors or lists on which the argument **FUN** operates and may be a collection of lists and/or vectors collected using `c()`. The argument **MoreArgs** refers to any additional arguments to **FUN** and by default equals **NULL**. The arguments must be in `list` mode, with a separate list for each argument.

The argument **SIMPLIFY** tells `mapply()` to attempt to simplify the result to a vector or matrix. The default value is **TRUE**. The argument **USE. NAMES** tells `mapply()` to use the names of the elements or, if the vector is of mode `character`, the characters themselves, as names for the output. By default, the value is **TRUE**. An example follows:

```
> set.seed( 382765 )

> a.mat = matrix( 1, 4, 4 )
> b.mat = matrix( runif( 9 ), 3, 3 )

> c.vec = 1:2

> mapply( det, list( a.mat, b.mat ) )
[1]  0.0000000 -0.3349038

> mapply( mean, c( list( a.mat, b.mat ), c.vec ) )
[1] 1.0000000 0.6208733 1.0000000 2.0000000

> mapply( mean, c( list( a.mat, b.mat ), list( c.vec ) ) )
[1] 1.0000000 0.6208733 1.5000000
```

Here, **det** finds the determinants of the elements, and **mean** finds the means of the elements.

Another example—using **MoreArgs**—follows:

```
> set.seed( 382765 )

> mapply( cor, c( list( a.mat, b.mat ), list( c.vec ) ),
                       list( y=1:4, y=1:3, y=3:4 ),
                       list( use="everything" ),
                       list( method="pearson" ) )
[[1]]
     [,1]
[1,]   NA
[2,]   NA
[3,]   NA
[4,]   NA

[[2]]
           [,1]
[1,]  0.1872769
[2,]  0.8836377
[3,] -0.4585219
```

```
[[3]]
[1] 1

Warning message:
In (function (x, y = NULL, use = "everything", method =
c("pearson",  :
  the standard deviation is zero
```

Here, the function is the correlation function and the arguments **y**, **use**, and **method** are supplied, each as a list. For the first matrix, four **y** values are given, so cor() is called four times since there are 16 elements in the matrix. For the second matrix, three **y** values are given so cor() is called three times. For the third matrix, two **y** values are given so cor() is only called once. The result is the three-element list. The NAs in the first element of the list result from the first matrix containing a single value only, so the correlations cannot be estimated for the first element.

You can find more information about `mapply()`by entering **?mapply** at the R prompt or by using the Help tab in R Studio.

The Function eapply()

The function eapply() applies a function to all objects in an environment and returns a list to the parent environment. The function takes five arguments, **env**, **FUN**, **..., all.names,** and **USE.NAMES**. The argument **env** is the name of the environment. The argument **FUN** is the function to be applied. The argument **...** gives any arguments to the function, separated by commas. The argument **all.names** is a logical variable indicating whether to include objects whose names begin with a period or not. The default value is FALSE. The argument **USE.NAMES** is a logical variable indicating whether the resultant list has names assigned to the elements or not. The default value is TRUE.

For example:

```
> nwenv = new.env()
> nwenv
<environment: 0x10b448d30>

> nwenv$a = 1:10
> nwenv$b = 11:20
> nwenv$c = rnorm( 100 )

> eapply( nwenv, sd )
$a
[1] 3.02765

$b
[1] 3.02765

$c
[1] 0.9947994

> ls( nwenv )
[1] "a" "b" "c"
```

Here, an environment is created and populated with three numeric objects. The function sd() (the function to find the standsrd deviation of the values in a numeric object) was applied to the three objects, and the resultant standard deviations were returned into .GlobalEnv as a three-element list.

More information about eapply() can be found by entering **?eapply** at the R prompt or by using the Help tab in R Studio.

The sweep() and scale() Functions

The `sweep()` function operates on arrays (including matrices and vectors that have been converted to matrices), and the `scale()` function operates on numeric matrixlike objects. The `sweep()` function sweeps out a margin(s) of an array (say, the columns of a matrix) with values (say, the column means) using a function (say, the subtraction operator). The `scale()` function by default centers and normalizes the columns of matrices by subtracting the mean and dividing by the standard deviation for each column.

The Function sweep()

The function `sweep()` takes the arguments **x**, **MARGIN**, **STATS**, **FUN**, **check.margin**, and **...**. The argument **x** is the array. The array can be of any atomic mode.

The argument **MARGIN** gives the margins over which the sweep is to take place. For a matrix, **MARGIN** equals **1**, **2**, or **1:2** (or **c(1,2)**). If **MARGIN** equals **1:2**, the entire matrix is swept, rather than the sweeping being done by column or row. For an array of more than two dimensions, **MARGIN** can be any subset of the margins, including all of the margins.

The argument **STATS** gives the value(s) to sweep with. For example, to use column means the function `apply()` can be applied; that is **apply(mat, 2, mean)** would work as a value for **STATS**, where **mat** is the matrix being swept. The value(s) for **STATS** cycle.

The argument **FUN** is the function to use. By default, **FUN** equals **"-"**, the subtraction operator, but **FUN** can be any function legal for the values of the array. For example, **paste** can be used with arrays of mode character.

The argument **check.margin** checks to see if the dimensions or length of **STATS** agrees with the dimensions given by **MARGIN**. If not, just a warning is given. The function does not stop but cycles the values in **STATS**. The default value is **TRUE**.

The argument **...** gives any extra arguments to the function **FUN**.

An example follows:

```
> a.mat = matrix( 1:8, 2, 4 )

> a.mat
     [,1] [,2] [,3] [,4]
[1,]    1    3    5    7
[2,]    2    4    6    8
```

```
> a.cent = sweep( a.mat, 2, apply( a.mat, 2, mean ) )

> a.cent
     [,1] [,2] [,3] [,4]
[1,] -0.5 -0.5 -0.5 -0.5
[2,]  0.5  0.5  0.5  0.5

> sweep( a.cent, 2, apply( a.mat, 2, sd ), "/" )
           [,1]       [,2]       [,3]       [,4]
[1,] -0.7071068 -0.7071068 -0.7071068 -0.7071068
[2,]  0.7071068  0.7071068  0.7071068  0.7071068
```

Since **MARGIN** is set equal to **2**, the function `mean()` takes the mean of each column, and the function `sd()` takes the standard deviation of each column. In the second statement, the mean of each column is subtracted from the elements in the column. The subtraction function is the default, so it does need not be entered. In the third statement, the centered elements in the columns are divided by the standard deviations of the columns.

Note that the function returns a matrix. You can find more information about `sweep()`by entering **?sweep** at the R prompt or by using the Help tab in R Studio.

The Function scale()

The function `scale()` is used to scale columns of a matrix—that is, to center the column to a specified center and to scale the column to a specified standard deviation. The function `scale()` takes three arguments: **x**, **center**, and **scale**. The argument **x** is a matrix or matrixlike numeric object (for example a data frame or time series).

The argument **center** can be either logical or a numeric vector of length equal to the number of columns in **x**. If set to **TRUE**, the column mean is subtracted from each element in a column. If set to a vector of numbers, then each number is subtracted from the elements in the number's corresponding column. If set equal to **FALSE**, nothing is subtracted. The default value is **TRUE**.

The argument **scale** can also be logical or a vector of numbers. If **scale** is set equal to **TRUE**, each centered (if centering has been done) element is divided by the standard deviation of the elements in the column, where **NA**s are ignored and the division is by n-1. If set equal to a vector of numbers, each (centered) element of a column is divided by the corresponding number in the vector. Dividing by zero will give an **NaN** but will not stop the execution. If **scale** is set equal to **FALSE**, no division is done. The default value is **TRUE**. An example follows:

```
> a.mat
     [,1] [,2] [,3] [,4]
[1,]    1    3    5    7
[2,]    2    4    6    8

> scale( a.mat )
           [,1]       [,2]       [,3]       [,4]
[1,] -0.7071068 -0.7071068 -0.7071068 -0.7071068
[2,]  0.7071068  0.7071068  0.7071068  0.7071068
attr(,"scaled:center")
[1] 1.5 3.5 5.5 7.5
attr(,"scaled:scale")
[1] 0.7071068 0.7071068 0.7071068 0.7071068

> a2.mat = matrix( c( 1:8, NA, 2 ), 2, 5 )

> a2.mat
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    3    5    7   NA
[2,]    2    4    6    8    2
```

```
> scale( a2.mat, center=rep( 3, 5 ), scale=rep( 4, 5 ) )
      [,1] [,2] [,3] [,4]  [,5]
[1,] -0.50 0.00 0.50 1.00    NA
[2,] -0.25 0.25 0.75 1.25 -0.25
attr(,"scaled:center")
[1] 3 3 3 3 3
attr(,"scaled:scale")
[1] 4 4 4 4 4
```

Note that `scale()` returns the scaled matrix, the values used to center the elements, and the values used to scale the elements.

For more information, enter **?scale** at the R prompt or use the Help tab in R Studio.

The Functions aggregate(), table(), tabulate(), and ftable()

Like the apply functions, the function `aggregate()` finds statistics for data groups. The functions `table()`, `tabulate()`, and `ftable()` create contingency tables out of data.

The Function aggregate()

The function `aggregate()` applies a function to the elements of an object based on the values of another object. The object to be operated on is either a time series, a data frame or an object that can be coerced to a data frame. The values of the other object must be a list with elements that can be interpretable as factors and, at the second level, must be of length equal to the rows of the data frame or time series. The function treats data frames and time series differently.

Data Frames

For data frames, the arguments are **x**, **by**, **FUN**, **...**, **simplify**, and **drop**. The argument **x** is a data frame. The argument **by** is an object of mode `list` consisting of elements that can be interpreted as factors. The elements of **by** are used to group the rows of **x**.

The argument **FUN** is the function to be applied and **...** are any extra arguments for that function. The argument **simplify** tells `aggregate()` whether to try to simplify the result to a vector or matrix. The default value is **TRUE**. The argument **drop** is a logical variable. If TRUE, unused combinations for the **by** factors are dropped. Starting with R 3.5.0 the default value is TRUE.

The result of `aggregate()` for a data frame is a data frame. An example follows:

```
>  x=rep( 1:2, 3 )
>  y1=1:6
>  y2=7:12

>  a.df=data.frame( y1, y2, x )

> a.df
  y1 y2 x
1  1  7 1
2  2  8 2
3  3  9 1
4  4 10 2
5  5 11 1
6  6 12 2

> aggregate( a.df[,1:2], by=list( x ), FUN=mean )
  Group.1 y1 y2
1       1  3  9
2       2  4 10
```

The function finds the means in each column for the two grouping values in x.

For data frames, a formula may be used to classify **x** rather than using the argument **by**. For the formula option, the arguments are **formula**, **data**, **FUN**, **...**, **subset**, and **na.action**. The argument **formula** takes the form **y~x**, where **y** is numeric and can have more than one column and **x** is a formula such as **x1** or **x1+x2**, where both **x1** and **x2** can be interpreted as factors.

The argument **data** gives the name of the data frame and must be included. The argument **FUN** is the function to be applied and **...** contains any extra arguments for **FUN**. The default value is sum. The argument **subset** gives the rows of the data frame on which to operate. The argument **na.action** gives the choice for how to handle missing values and is a character string. The default value is **"na.omit"**, which tell `aggregate()` to omit missing values. An example follows:

```
> a.df
  y1 y2 x
1  1  7 1
2  2  8 2
3  3  9 1
4  4 10 2
5  5 11 1
6  6 12 2

> aggregate( cbind( y1, y2 )~x, data=a.df, sum, subset=1:3 )
  x y1 y2
1 1  4 16
2 2  2  8
```

The first three rows of y1 and y2 are summed based on the value of x.

Note that the **by** variable must be a list while the right side of a formula cannot be a list.

Time Series

Time series have both a frequency and a period. In R, the frequency is the inverse of the period and vice versa. For example, a year can be the period of interest. Then, the months have a frequency of 12 while having subperiods of 1/12.

For time series, the arguments are **x**, **nfrequency**, **FUN**, **ndeltat**, **ts.eps**, and **...**. The argument **x** must be a time series. The argument **nfrequency** is the number of subperiods for each period after **FUN** has operated on the time series. The value must divide evenly into the original time series frequency. For a monthly time series, aggregating to a quarter can be done by setting nfrequency to four. The argument equals **1** by default. (The original time series frequency divided by **nfrequency** gives the number of elements that are grouped together—on which **FUN** operates.)

The argument **FUN** is the function to be applied and **...** gives any extra arguments to **FUN**. The argument **...** is at the end of the argument list. The function **FUN** must be legal for the values of the time series and is by default **sum**.

The argument **ndeltat** tells `aggregate()` the length of the subperiods for the output and equals **1** by default. The argument is the value of one divided by **nfequency**. The product of the frequency of the original time series and **ndelta**t must be an integer.

Either **nfrequency** or **ndeltat** can be set but not both. The product of **nfrequency** and the inverse of **ndeltat** is the frequency of the original time series, or its inverse if **nfrequency** is less than one.

The argument **ts.eps** gives the tolerance for accepting that **nfrequency** divides evenly into the frequency of the time series. By default, **ts.eps** equals **getOption("ts.eps")**, which value can be found by entering **options("ts.eps")** at the R prompt. The value is numeric and can be set manually.

An example follows:

```
> a.ts=ts( cbind( 1:12, 11:22 ), start=c( 1, 1 ), freq=4 )

> a.ts
     Series 1 Series 2
1 Q1        1       11
1 Q2        2       12
1 Q3        3       13
1 Q4        4       14
2 Q1        5       15
2 Q2        6       16
2 Q3        7       17
2 Q4        8       18
3 Q1        9       19
3 Q2       10       20
3 Q3       11       21
3 Q4       12       22

> aggregate( a.ts, nfreq=2 )
Time Series:
Start = c(1, 1)
End = c(3, 2)
Frequency = 2
    Series 1 Series 2
1.0        3       23
1.5        7       27
2.0       11       31
2.5       15       35
3.0       19       39
3.5       23       43
```

```
> aggregate( a.ts, ndelt=1/2 )
Time Series:
Start = c(1, 1)
End = c(3, 2)
Frequency = 2
    Series 1 Series 2
1.0        3       23
1.5        7       27
2.0       11       31
2.5       15       35
3.0       19       39
3.5       23       43

> aggregate( a.ts, nfreq=1/2 )
Time Series:
Start = 1
End = 1
Frequency = 0.5
  Series 1 Series 2
1       36      116

> aggregate( a.ts, ndelt=2 )
Time Series:
Start = 1
End = 1
Frequency = 0.5
  Series 1 Series 2
1       36      116
```

Note that **nfreq** can be less than one but must give an integer if multiplied by **freq**. In the example with **nfreq=1/2**, the first eight rows are summed, but the last four rows are ignored.

You can find more information about `aggregate()` by entering **?aggregate** at the R prompt or by using the Help tab in R Studio.

The Functions table(), as.table(), and is.table()

There are three functions associated with creating tables using `table()`. The function `table()` creates a contingency table from atomic data or some lists. The data must be able to be interpreted as factors. The result has class table. The function `as.table()` attempts to coerce an object to class table. The function `is.table()` tests if an object is of class `table`.

The arguments to `table()` are **...**, **exclude**, **useNA**, **dnn**, and **deparse.level**.

The argument **...** refers to the object(s) that are to be cross-classified. The objects are separated by commas and, for atomic objects, must have same length. For list objects, the second level elements must all have the same length and be atomic. Atomic and list objects cannot be combined in a call to `table()`.

The argument **exclude** gives values to be excluded from the contingency table. By default, **exclude** equals **if(useNA==“no”) c(NA, NaA)**, which tells `table()` not to set a level for missing values or illegal values—such as one divided by zero—if the argument **useNA** equals **“no”**.

The argument `useNA` is a character argument and can take on the value **“no”**, **“ifany”**, or **“always”**. For **“no”**, no level is set for missing values. For **“ifany”**, a level is set if missing values are present. For **“always”**, a level for missing values is always set. The default level is **“no”**.

The argument **dnn** is a list argument and gives dimension names for the contingency table. The default value is list.names(...). The function `list.names()` is defined in `table()` and gives the names of the dimensions being tabulated.

The argument **deparse.level** is an integer argument that can take on the values of **0**, **1**, or **2**. The argument controls list.names() if **dnn** is not given. For **0**, no names are given. For **1**, the column names are used.

For **2**, column names are deparsed. The default value is **1**. An example follows:

```
> set.seed(203846)

> a1.samp=sample( 3, 100, replace=T )

> a2.samp=sample( 3, 100, replace=T )

> table( a1.samp, a2.samp )
       a2.samp
a1.samp  1  2  3
      1 12 10 14
      2 13  9  9
      3 15  8 10

> a2.samp[10]=NA

> table( a1.samp, a2.samp )
       a2.samp
a1.samp  1  2  3
      1 12 10 14
      2 12  9  9
      3 15  8 10

> table( a1.samp, a2.samp, useNA="ifany" )
       a2.samp
a1.samp  1  2  3 <NA>
      1 12 10 14    0
      2 12  9  9    1
      3 15  8 10    0
```

Note that the second table does not include the missing value, but the third does.

The function as.table() takes the arguments **x** and **...**. The argument **x** is the object to be coerced to the table class. The argument must be of mode numeric. The argument**...** provides any arguments for lower-level functions.

The function is.table() takes the argument **x** and returns **TRUE** if **x** is of class table and **FALSE** if not.

You can find more information about table(), as.table(), and is.table() by entering **?table()** at the R prompt or by using the Help tab in R Studio.

The Function tabulate()

The function tabulate() coerces numeric or factor objects to vectors and bins the result. The arguments are **bin** and **nbins**. The argument **bin** is the object to be binned. If the object is not an integer or factor object, then the elements are rounded down to integers. The resulting integers must be positive. If an illegal element is present, the element is ignored.

The argument **nbins** gives the largest integer to be binned and by default equals **max(1, bin, na.rm=T)**—that is, the largest value in **bin**, assuming the largest value in **bin** is larger than one. By default, NAs are removed.

If **nbins** is smaller than the largest value in **bin**, then only those values with a value less than or equal **nbins** are binned. All of the integers between one and **nbins** are binned even if there are zero elements in a given bin. The function creates a vector without labels. The bins always start with one. An example follows:

```
> tabulate( c( -3.5, .9, 1, 4, 5.6, 5.4, 4, 1, 3) )
[1] 2 0 1 2 2

> tabulate( c( -3.5, .9, 1, 4, 5.6, 5.4, 4, 1, 3 ), nbins=3 )
[1] 2 0 1
```

In the example, there are two ones, zero twos, one three, two fours, and two fives in the reduced object.

The function `tabulate()` is good when all of the bins, including those with zero elements, are needed. You can find more information about `tabulate()`by entering **?tabulate** at the R prompt or by using the Help tab in R Studio.

The Function ftable()

The function `ftable()` creates a matrix out of a contingency table—that is, a matrix that is a flat table. The arguments are **...**, **exclude**, **row.vars**, and **col.vars**. The argument **...** can be objects that can be coerced to a vector and that can be interpreted as factors, separated by commas. The argument can also be a list whose elements can be interpreted as factors, or the argument can be of class `table` or `ftable`.

The argument **exclude** gives the values to be excluded when building the flat table. By default, **exclude** equals **c(NA, NaN)**.

The arguments **row.vars** and **col.vars** give the dimensions to put in the rows and columns. The values can go from one to the number of dimensions in the table—in other words, a table with three dimensions can have **row.vars** and **col.vars** equal to **1:2** and **3**; or **2:1** and **3**; or **1** and **3**; or **c(3,1)** and **2**; and so forth. An example follows:

```
> a.list = list( 1:2, 3:4, 5:6 )

> ftable( a.list )
        x.3 5 6
x.1 x.2
1   3       1 0
    4       0 0
2   3       0 0
    4       0 1
```

```
> a1 = 1:2
> a2 = 3:4
> a3 = 5:6

> ftable( a1, a2, a3, row.vars=3, col.vars=2:1 )
   a2 3   4
   a1 1 2 1 2
a3
5     1 0 0 0
6     0 0 0 1

> a.table = table( 1:2, 3:4, 5:6 )

> ftable( a.table, row.vars=2, col.vars=3 )
   5 6

3  1 0
4  0 1
```

In these examples, the two observations are (1,3,5) and (2,4,6).

You can find more information about `ftable()` by entering **?ftable** at the R prompt or by using the Help tab in R Studio.

Some Character String Functions

There are a number of functions for searching for patterns in character strings and for replacing parts of strings with other strings based on matching. This section covers the grep functions, the sub functions, the regex functions, the str functions, and the character case transformation functions.

The grep Functions

The grep() and grepl() functions search for matches to a pattern in a vector of character strings. The function grep() returns either the index or the value of those strings that contain the pattern. The function grepl() returns a logical vector of the same length as the character vector with elements equal to TRUE if there is a match, and FALSE if there is not a match, for each element of the character vector.

The arguments of grep() are **pattern**, **x**, **ignore.case**, **perl**, **value**, **fixed**, and **useBytes**. The argument **pattern** is a character string or an object that can be coerced to a character string by using as.character(). If the argument contains more than one element, only the first one is used. The argument **x** is the character vector in which to look for the matches. The argument **ignore.case** tells grep() to ignore case in doing the matching if set equal to TRUE. The default value is FALSE.

The arguments **perl** and **fixed** tell grep() what type of matching to do. (See the help page for **regex** for more information.) Both arguments are FALSE by default. The argument **value** tells grep() to return the value of the element if set to TRUE and the index of the element if set to FALSE. The default value is FALSE. The argument **useBytes**, if set to TRUE, tells grep() to match byte-wise rather than character-wise. The default value is FALSE. The argument **inverse**, if set equal to TRUE, tells grep() to return the elements that do not contain matches rather than those that do. The default value is FALSE.

An example:

```
> ab.char=c( "achar1", "achar2", "achar3" )

> ab.char
[1] "achar1" "achar2" "achar3"

> grep( "achar", ab.char )
[1] 1 2 3
```

```
> grep( "1", ab.char, value=T )
[1] "achar1"

> grep( "Achar", ab.char )
integer(0)

> grep( "Achar", ab.char, ignore.case=T )
[1] 1 2 3

> grep( "Achar", ab.char, ignore.case=T, invert=T )
integer(0)
```

The function grepl() takes the same arguments as grep() except that there are no arguments **value** or **invert**. The function returns a logical vector, for example:

```
> grepl( "1", ab.char )
[1]  TRUE FALSE FALSE

> grepl( "Achar", ab.char )
[1] FALSE FALSE FALSE
```

The functions agrep() and agrepl() are similar to grep() and grepl(), except that agrep() and agrepl() do “fuzzy” matching. For example:

```
> grepl( "Achar", ab.char )
[1] FALSE FALSE FALSE

> agrepl( "Achar", ab.char )
[1] TRUE TRUE TRUE
```

See the help page for agrep() for more information on how the matching can be done.

The function grepRaw() does pattern matching for raw vectors. The function takes the arguments **pattern**, **x**, **offset**, **ignore.case**, **value**, **fixed**, **all**, and **invert**.

The argument **pattern** is the pattern to be matched and can be a raw vector or a single character string. The argument **x** is also a raw vector or a single character string and is the object in which to search for the pattern. In grepRaw(), before the search, the character strings are converted to raw vectors using the function charToRaw().

The argument **offset** gives the index of the raw vector at which to start searching. The value must be able to be coerced to a positive integer. If the value is an object of length greater than one, only the first element is used. The default value for **offset** is 1L.

The argument **ignore.case**, if set equal to TRUE, tells grepRaw() to match both capital letters and lower case letters given a letter of either case. The default value is FALSE.

The argument **value**, if set equal to TRUE, returns the first raw vector containing the match or a list of the raw vectors containing the matches, depending on whether the argument **all** is FALSE or TRUE. If **value** is FALSE, either the index of the first element of the first match, or the indices of the first elements of all of the matches, is(are) returned, depending on the value of the argument **all**—FALSE or TRUE. The default value of **value** is FALSE.

The argument **all** tells grepRaw() to just return the first match if set equal to FALSE and all matches if set equal to TRUE. The default value is FALSE. The arguments **fixed** and **invert** are as defined for grep() and by default are FALSE.

An example:

```
> a=charToRaw( "abc123" )
 > a
[1] 61 62 63 31 32 33

> grepRaw( "b", a )
[1] 2
```

```
> grepRaw( "b", a, value=T )
[1] 62

> grepRaw( "B", a, value=T, ignore.case=T )
[1] 62

> grepRaw( "ab", "abab" )
[1] 1

> grepRaw( "ab", "abab", all=T )
[1] 1 3

> grepRaw( "ab", "abab", value=T, all=T )
[[1]]
[1] 61 62

[[2]]
[1] 61 62

> grepRaw( "ab", "Abab", value=T, all=T )
[[1]]
[1] 61 62

> grepRaw( "ab", "Abab", value=T, all=T, ignore.case=T )
[[1]]
[1] 41 62

[[2]]
[1] 61 62
```

The functions sub() and gsub() replace a new string for a substring in the element(s) of an object that can be coerced to a character vector. The arguments to both functions are **pattern**, **replacement**, **x**, **ignore.case**, **perl**, **fixed**, and **useBytes**. The only new argument is **replacement**, the replacement value. The replacement value must be an object that can be coerced to a character string. If the replacement object has more than one element, only the first element is used, and a warning is given. The function sub() replaces the first occurrence of the pattern in each element of **x**. The function gsub() replaces all occurrences of the pattern.

For example:

```
> sub( "b1", "c", c( "b1b2b1", "cb1" ) )
[1] "cb2b1" "cc"

> gsub( "b1", "c", c( "b1b2b1", "cb1" ) )
[1] "cb2c" "cc"
```

The functions regexpr(), gregexpr(), and regexec() return the location and length of a string within a character vector, plus some other attributes such as type of expression. For all three of the functions, a list is returned. A minus one is returned if no match is found. The arguments to the three functions are **pattern**, **text**, **ignore.case**, **perl**, **fixed**, and **useBytes**. Here, **text** is the object in which to search for the pattern. The other arguments are as described previously.

The function regexpr() finds the first occurrence of the pattern for each element of **text** and returns a vector and some attributes. The vector is a vector of integers, where for each element in **text**, the integer is the position of the first occurrence of the pattern in the element. If the pattern is not in the element, a minus one is used.

The first attribute of the result is "match.length"—a vector of integers which contains the number of characters or bytes (depending on whether **useBytes** is FALSE or TRUE) in the first match of the pattern. Again, if there is no match, minus one is used. Two other possible attributes are "index. type" and "useBytes."

To separate out the vector from the attributes, you can use the function as.vector() on the result. To access the attributes, you can use the function attr().

For example:

```
> a=regexpr( "ab", c( "ababab", "ba" ) )

> a
[1]  1 -1
attr(,"match.length")
[1]  2 -1
attr(,"index.type")
[1] "chars"
attr(,"useBytes")
[1] TRUE

> as.vector( a )
[1]  1 -1

> attr( a, "match.length" )
[1]  2 -1
```

The function gregexpr() finds all matches to the argument **pattern** in each element of **text**. The function takes the same arguments as regexpr() and returns a list of the same length as **text**. The first element of the list contains the information for the first element of **text**; the second information about the second; and so forth. The structure of each element of the list is structured like the output from regexpr() except the reference is to all matches in the element rather than for the first match in each element.

For example:

```
> ag=gregexpr( "ab", c( "ababab", "ba" ) )

> ag
[[1]]
[1] 1 3 5
```

```
attr(,"match.length")
[1] 2 2 2
attr(,"index.type")
[1] "chars"
attr(,"useBytes")
[1] TRUE

[[2]]
[1] -1
attr(,"match.length")
[1] -1
attr(,"index.type")
[1] "chars"
attr(,"useBytes")
[1] TRUE

> as.vector( ag[[1]] )
[1] 1 3 5

> as.vector( ag[[2]] )
[1] -1
```

The function regexec() is regexpr() with output in the form of gregexpr().

For more information on grep(), grepl(), sub(), gsub(), regexpr(), gregexpr(), and regexec() enter **?grep** at the R prompt or use the Help tab in R Studio. For more information about agrep() and grepRaw(), enter **?agrep** and **?grepRaw** at the R prompt or use the Help tab in R Studio.

Functions to Manipulate Case in Character Strings

Three functions that can be used to change the case of a character string are tolower(), toupper(), and chartr(). The functions tolower() and toupper() take one argument, **x**, which can be any vector that can be coerced to character by using as.character(). The functions change the case of the entire vector either to lower or upper case. Characters that are not letters are not changed.

For example:

```
> tolower( c( "Jane Doe", "John Doe" ) )
[1] "jane doe" "john doe"
```

```
> toupper( c( "Jane Doe", "John Doe" ) )
[1] "JANE DOE" "JOHN DOE"
```

The function chartr() changes characters in a vector, **x**, to other characters. The function takes three arguments, **old**, **new** and **x**. The arguments **old** and **new** must be character strings and of the same length. The characters to be replaced make up **old**, while the replacement characters are in **new**, where there is a one to one transformation between the two. Each character in the string is evaluated separately. Characters can be referred to by a range.

For example:

```
> chartr( "ao", "oa", c( "Jane Doe", "John Doe" ) )
[1] "Jone Dae" "Jahn Dae"
```

```
> chartr( "a-e", "ABCDE", c( "Jane Doe", "John Doe" ) )
[1] "JAnE DoE" "John DoE"
```

More information about tolower(), toupper(), and chartr() can be found by entering **?tolower** at the R prompt or by using the "Help" tab in R Studio.

The Functions substr( ), substring( ), and strsplit( )

The functions substr(), substring(), and strsplit() work with strings by specifying where on the string to operate. The function substring() takes three arguments, **x**, **start**, and **stop**. The argument **x** is a character vector; the argument **start** tells substr() the how far into the character string to go before selecting or changing the sub string; the argument **stop** tells substr() where to stop. Both values should be positive integers. Either of the integers can be larger than the number of characters in a string. Neither **start** nor **stop** has default values.

For example:

```
> substr( c( "Jane Doe", "John Doe", "Ms. X" ), 2, 7 )
[1] "ane Do" "ohn Do" "s. X"

> substr( c( "Jane Doe", "John Doe", "Ms. X" ), 6, 7 )
[1] "Do" "Do" ""

> a.str=c( "Jane Doe", "John Doe", "Ms. X" )

> substr( a.str, 6, 7 ) = "soA"

> a.str
[1] "Jane soe" "John soe" "Ms. X"
```

In the first part of the example, substr() operates on the second through seventh characters in each element of the vector. In the second part, substr() operates on the sixth through seventh characters in each element. Note that the third element only has five characters. In the third part, only two characters are replaced, characters six and seven.

The function substring() performs much like substr(), except that the three arguments are **text**, **first**, and **last**. **last** has the default value of 1000000L, so need not be specified.

Using “a.str” from the above example

```
> a.str
[1] "Jane Doe" "John Doe" "Ms. X"

> substring( a.str, 2 ) = "osa"

> a.str
[1] "Josa Doe" "Josa Doe" "MosaX"
```

The function strsplit() splits the elements of a character vector into a list of smaller vectors based on a string or an object that can be coerced to a string. The function takes five arguments, **x**, **split**, **fixed**, **perl**, and **useBytes**. The arguments **fixed**, **perl**, and **useBytes** are as described previously and on the help page. The argument **x** is the object to be split and must be a character vector. The argument **split** is the string used for splitting. The value(s) in string are not included in the split. For splitting on periods, use the string "[.]" rather than ".". To split out the string into individual characters set **string** to "", NULL, or character(0).

For example:

```
> strsplit( "a.b.b", "b." )
[[1]]
[1] "a." "b"

> strsplit( "a.b.b", "." )
[[1]]
[1] "" "" "" "" ""

> strsplit( "a.b.b", "[.]" )
[[1]]
[1] "a" "b" "b"

> strsplit( c( "a.b.b", "d.f.d" ), "" )
[[1]]
[1] "a" "." "b" "." "b"

[[2]]
[1] "d" "." "f" "." "d"
```

More information about substr() and substring() can be found by entering **?substr** at the R prompt or by using the Help tab in R Studio. For strsplit(), enter **?strsplit** or use the Help tab.