CHAPTER 3

Assignments and Operators

R works with objects. Objects can include vectors, matrices, functions, the results from a function, or a number of other kinds of objects. Objects make working with information easier. This chapter covers assigning names to objects, listing and removing objects, and object operations. Part II (Chapters 4 and 5) covers the possible forms of objects.

Some objects come with the packages in R. Other objects are user created. User-created objects have names that are assigned by the user. Knowing how to create, list, and remove user-created objects is basic to R. In R Studio, the user-created objects are listed in the upper right window under the Environment tab.

Types of Assignment

Names in R must begin with a letter or a period, cannot have breaks, and can contain letters, numeric digits, periods, and underscores. The names that begin with a period are hidden and are used by R for startup defaults, the random seed, and other such things. The indexing symbols **[]**, **[[]]**, **$**, and **@** have special meanings with regard to R names, as explained in the "Subscripting Operators" section of this chapter.

M. Tollefson, *R Quick Syntax Reference*, https://doi.org/10.1007/978-1-4842-4405-0_3

R originally used five types of assignment, four of which are still current. The four types are

```
a <- b,
```

which assigns **b** to **a**,

```
a -> b,
```

which assigns **a** to **b**,

```
a <<- b,
```

which assigns **b** to **a** and can be used inside a function to bring the assignment up to the workspace level, and

```
a ->> b,
```

which assigns **a** to **b** and brings an assignment in a function up to the workspace level.

The developers at R have also included the more standard

```
a = b,
```

which assigns **b** to **a**. Using the equal sign for assignment is considered poor practice in R, but we have never had a problem using it. While any of the types of assignment can be used, the use of the equal sign is easiest to type.

When R makes an assignment, the name is automatically saved in the workspace. Note that no warning is given if the assigned name already exists. The assignment will overwrite the object in the workspace with the assigned object.

R is interesting in that a function of an object can be assigned to the original object. For example,

```
a = 2*a,
```

where the object **a** is replaced by the original **a** times two.

For more information about assignment operators, enter **??"Assignment Operators"** at the R prompt or in R Studio use the Help tab in the lower right window.

Example of Three Types of Assignment

An example of some of the types of assignment follows. Three objects are created: **abc**, **bcd**, and **cde**. You create the objects by assigning sequences to the objects. The sequences are generated when you put a colon between two integers, which creates a sequence of integers starting with the first integer and ending with the second integer.

To show that the objects actually contain the assigned sequence, the contents of the three objects are displayed as follows. Note that entering the name of an object at the R prompt will always display the contents of the object. The **[1]** refers to the first element of the objects.

```
> abc = 1:10

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

> bcd <- 11:20

> bcd
 [1] 11 12 13 14 15 16 17 18 19 20

> 21:30 -> cde

> cde
 [1] 21 22 23 24 25 26 27 28 29 30
```

As you can see, the assignment operators <- and = give the same result. The assignment operator -> works in the opposite direction.

Listing and Removing Objects in R and R Studio

To see the objects present in the workspace, it is easier to use R Studio rather than R - look under the Environment tab in the right upper window. R has the function `ls()` to list the workspace objects.

Entering **ls()** at the R prompt for the preceding example gives

```
> ls()
[1] "abc" "bcd" "cde"
>,
```

which are the three objects created previously.

Although functions are covered in detail in Part III, one interesting property of functions to note here is they can have arguments that the user enters. Two of the possible arguments for `ls()` are **pattern** and **all.names**.

The first argument is entered as **pattern** = ***"a string"***, where *"a string"* is any part of an object name. For example, in the preceding workspace, searching for those objects containing **bc** in the name gives **abc** and **bcd**, that is

```
> ls( pattern="bc" )
[1] "abc" "bcd"
```

The argument **pattern** can be reduced to **pat**, as in **ls(pat="bc")**. The shortening of arguments of functions is a property of R. All arguments in R can be reduced to the shortest unique form, but they are usually given in the full form in manuals.

The second argument is **all.names**, which can equal **TRUE** or **FALSE**. If set to **TRUE**, the **all.names** argument instructs R to list all of the files in the workspace, including those that begin with a period. **FALSE** is the

default value and does not need to be entered. For the previous example workspace, setting **all.names** equal to **TRUE** gives

```
> ls( all.n=T )
[1] ".commander.done" ".First"   ".Random.seed"   ".Traceback"
[5] "abc"             "bcd"      "cde"
.
```

The **[1]** refers to ".commander.done" since ".commander.done" is the first element of the vector, and the **[5]** refers to "abc" since "abc" is the fifth element of the vector. (In R, if the elements of a vector have not been given a name, the convention for listing the elements is to show the index of the first element in each line of the lines of listed elements.)

The function `rm()` can be used to remove objects from the workspace. For `rm()`, the names of the objects to be deleted are separated by commas. For example,

```
rm( a, b, c )
```

will remove objects **a**, **b**, and **c**. To remove all objects,

```
rm( list=ls() )
```

works. You remove S4 classes by using removeClass().

In R Studio, objects can be removed under the grid option for listing the environmental objects. To the right side of the menu under the Environment tab is an icon that says List. Click on the icon and choose Grid instead of List. In the resulting grid, check the boxes to the left of the objects to be removed. Then, click on the little broom in the middle of the menu. You will be asked if you really want to delete the checked objects.

For more information about `ls()` or `rm()`, enter **?ls** or **?rm** at the R prompt or, in R Studio use the Help tab in the lower right window.

Operators

Operators operate on objects. Operators can be logical, arithmetic, matrix, relational, or subscripting, or they may have a special meaning. Each of the types of operators is described here.

For operators, *elementwise* refers to performing the operation on each element of an object or paired elements for two objects. If two objects do not have the same dimensions, the operator will often cycle the smaller object against the larger object. The cycling proceeds through each dimension. For example, for matrices, the first dimension is the rows and the second dimension is the columns, so the cycling is down rows starting with the first column.

The letters **NA** are used to indicate that an element is missing data. Most operators have rules for dealing with missing data and may return an **NA** if data is missing.

CRAN gives a help page of information about operation precedence. Enter **??"Operator Syntax and Precedence"** at the R prompt to see the page or use the Help tab in R Studio.

Logical Operators and Functions

Logical operators and functions return the values **TRUE**, **FALSE**, or **NA**, where **NA** refers to a missing value. The logical operators are the **not** operator, two **or** operators, and two **and** operators. The functions **xor()**, **isTRUE()**, **isFALSE()**, **any()**, and **all()** (which are functions that operate on logical objects) also return logical values. For logical operators, if the two objects do not have the same dimensions, the number of elements in the larger object must be a multiple of the number of elements in the smaller object for cycling to occur. The logical operators and five logical functions are listed in Table 3-1.

Table 3-1. *The Logical Operators and Functions*

Operator	Operation	Description
!	not	negation operator—e.g., !a
|	or	elementwise **or** operator—e.g., a|b
||	Or	**or** operator, just evaluates the first elements in the objects—e.g., a||b
&	and	elementwise **and** operator—e.g., a&b
&&	And	**and** operator, just evaluates the first elements in the objects—e.g., a&&b
xor()	exclusive or	**exclusive or** function—e.g., xor(a,b)
isTRUE()	logical test	returns **TRUE** if the argument contains only one value and the value is true, otherwise returns **FALSE**—e.g., isTRUE(a)
isFALSE()	logical test	returns **TRUE** if the argument contains only one value and the value is false, otherwise returns **FALSE**—e.g., isFALSE(a),
any()	logical test	returns **TRUE** if **TRUE** is present in a logical object— e.g., any(a)
all()	logical test	returns **TRUE** if **TRUE** is the only value in a logical object— e.g., all(a)

The logical operators operate on objects that are logical, numeric, or raw. When a numeric object is coerced to logical, all of the nonzero values are set to **TRUE,** and the zero values are set to **FALSE**. For raw vectors, the operators are applied bitwise.

The negation operator changes **TRUE** to **FALSE** and **FALSE** to **TRUE** in a logical object. An **NA** remains an **NA**.

The operator | compares the two objects elementwise and, for each pair of elements, returns **TRUE** if **TRUE** is present, **FALSE** if no **TRUE** or **NA** is present, and **NA** if any **NA** is present. The operator || compares the first element of the first object to the first element of the second object and returns **TRUE** if both elements are **TRUE**, **FALSE** if both are **FALSE** and **NA** if either element is **NA**.

The operator **&** compares two objects elementwise and, for each pair of elements, returns **TRUE** if both elements are **TRUE**, **FALSE** if **FALSE** is present, and **NA** if both elements are **NA**. The operator **&&** compares the first element of the first object to the first element of the second object and returns **TRUE** if the first elements are both **TRUE**, **FALSE** if **FALSE** is present, and **NA** if both elements are **NA**.

The `xor()` function compares objects elementwise and returns **TRUE** if the paired elements are different and **FALSE** if the paired elements are the same, unless an **NA** is present. If an **NA** is present, the test returns **NA**.

For a logical vector or a vector that can be coerced to logical, the function `any()` will return **TRUE** if any of the elements are **TRUE**, **FALSE** if no **TRUE** or **NA** is present, and **NA** if no **TRUE** is present but an **NA** is.

For a logical vector or a vector that can be coerced to logical, the function `all()` will return **TRUE** if all of the elements are **TRUE,** otherwise **FALSE** if a **FALSE** is present and **NA** if not.

The functions **isTRUE()** and **isFALSE()** only evaluate single element objects or expressions. If more than one element is present, the function will give an error. The function **isTRUE()** returns TRUE if the value is TRUE and FALSE if otherwise. The function **isFALSE()** returns TRUE if the value is FALSE and FALSE otherwise.

For more information about the logical operators and the functions **isTRUE()** and **isFALSE()**, the CRAN help pages for logical operators can be found by entering **??"logical operators"** at the R prompt or by using the Help tab in R Studio. The help page for `any() and all()` can be accessed by entering **?any** or **?all** at the R prompt or by using the Help tab in R Studio.

Arithmetic Operators

Arithmetic operators can have numeric operands or operands that can be coerced to numeric. For example, for logical objects, **TRUE** coerces to **1** and **FALSE** coerces to **0**. For some types of objects, specific operators have a different meaning, but those types of objects will not be covered in this chapter.

Arithmetic expressions are evaluated elementwise. If the number of elements is not the same between the objects in an expression, the smaller object cycles through the larger one until the end of the larger one. The numbers of elements in the larger object do not have to be a multiple of the smaller object for cycling. Expressions are evaluated from left to right, under the rules of precedence.

The arithmetic operators are the standard * for multiplication, / for division, + for addition, and - for subtraction. The exponentiation symbol is ^. The operator **%%** gives the modulus of the first argument with respect to the second argument. The operator **%/%** performs integer division. Expressions can be grouped using parentheses, for example **(a+b)/c**. Table 3-2 lists the arithmetic operators.

Table 3-2. *Arithmetic Operators*

Operator	Operation	Example
*	multiplication	a*b
/	division	a/b
+	addition	a+b
-	subtraction	a-b
^	exponentiation	a^b
%%	modulus	a%%b
%/%	integer division	a%/%b

For more information, the CRAN help pages for arithmetic operators can be found by entering **??"arithmetic operators"** at the R prompt or by using the Help tab in R Studio. (At the time of writing, this help page was not available using the above but can be brought up by using ?"+" at the R prompt or + in the R Studio Help tab search box.)

Matrix Operators and Functions

R provides operators and functions to manipulate matrices. A list of some matrix operators and functions can be found in Table 3-3.

The matrix multiplication operator is **%*%**. R will return an error if the two matrices do not conform.

For two arrays (arrays include vectors and matrices), **%o%**, or outer(), gives the outer product of the arrays.

For two arrays, **%x%**, or kronecker(), gives the kronecker product of the arrays.

To transpose a matrix, use the function `t()`, for example, t(a).

To get the cross product of one matrix with another (or the original matrix), use either the function `crossprod()` or the function `tcrossprod()`. If **a** and **b** are conforming matrices, then

```
crossprod(a) = t(a)%*%a,

tcrossprod(a) = a%*%t(a),

crossprod(a,b) = t(a)%*%b,

tcrossprod(a,b) = a%*%t(b).
```

To find the inverse of a nonsingular square matrix, use the function `solve()`, for example, solve(a). The function `solve()` also can solve the linear equation

```
Xa=b,
```

for **a**, where **X** is a nonsingular square matrix and **b** has the same number of rows as **X**. The syntax is solve(X,b).

To find the determinant of a square matrix use det(X), where **X** is a square matrix.

Table 3-3. *Matrix Operators and Functions*

Operator / Function	Operation	Example
%*%	matrix multiplication	a%*%b
%o% or **outer()**	outer product of two vectors, matrices, or arrays	a%*%b, outer(a,b)
%x% or **kronecker()**	kronecker product of a matrix (or array)	a%x%b, kronecker(a,b)
t()	transpose of a matrix	t(a)
crossprod() or **tcrossprod()**	crossproduct of a matrix or two matrices	crossprod(a) or crossprod(a,b) or tcrossprod(a) or tcrossprod(a,b)
diag()	diagonal of a matrix or a diagonal matrix	diag(a), **a** is a matrix or diag(a), **a** is a vector
solve()	inverse of a matrix or solution to **Xa=b**	solve(a), solve(X,b)
det()	determinant of a square matrix	det(a)

To create a diagonal matrix or obtain the diagonal of a matrix, use the function `diag()`. If **a** is a vector, diag(a) will return a diagonal matrix with the diagonal equal to the **a**. For example:

```
> a = 1:2
> a
[1] 1 2
```

```
> diag(a)
     [,1] [,2]
[1,]    1    0
[2,]    0    2
```

If **a** is a matrix, diag(a) will return the diagonal elements of the matrix, even if the matrix is not square. For example:

```
> a = matrix(1:6,2,3)
```

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

```
> diag(a)
[1] 1 4
```

For more information, the CRAN help page for matrix multiplication can be found by entering **??"matrix multiplication"** at the R prompt. For the six functions, entering **?*name***, where *name* is the name of the function, brings up the help page for the function. You can also use the R Studio Help tab. Enter ?"%" at the R prompt or % in The R Studio Help tab to see the % operators.

Relational Operators

Relational operators are used in logical tests. The six relational operators are == for equal to, **!**= for not equal to, < for less than, <= for less than or equal to, > for greater than, and >= for greater than or equal to. The list of logical operators can be found in Table 3-4.

Table 3-4. *Logical Operators*

Operator	Operation	Example
==	equals	a==9
!=	not equal	a!=9
>	greater than	a>9
>=	greater than or equal to	a>=9
<	less than	a<9
<=	less than or equal to	a<=9

Note that the **equal to** relational operator is ==, not =. A common mistake is to enter = for == in a logical expression. R will return an error for =.

As with arithmetic operators, logical expressions can be grouped using parentheses. For example,

```
( ( a>0 & b>0 ) & ( a<5 & b<5 ) )
```

is a logical expression and can be assigned a name.

The CRAN help page for relational operators can be found by entering **??"relational operators"** at the R prompt or == in the Help tab search box in R Studio.

Subscripting Operators

Many objects in R have more than one element. Subscripting is used to access specific elements of an object. Vectors, matrices, arrays, lists, and slots can be subscripted. In S3, single square brackets (**[]**), double square brackets (**[[]]**), and dollar signs (**$**) are used. For S4 objects, the *at* symbol (**@**) is used for subscripting. None are used elsewhere.

Vectors

For vectors, except list vectors, using single square brackets is usually appropriate. Double square brackets can also be used, but they can only access a single element of the vector at a time. Within single square brackets, there may be a logical expression or a set of indices. For example:

```
a[ 3:7 ]  or  a[ a>3 ]
```

The first expression results in the third through seventh elements of **a**. The second expression results in those elements of **a** that are greater than three.

If indices are given a negative sign, those indices are not included. For example,

```
a[ -2:-6 ]
```

would return the object **a** with elements two through six removed.

An object can be subsetted in one set of square brackets and subsetted again in another set of square brackets. For example:

```
a[ 1:10 ][ b>3 ],
```

where the length of **a** is greater than or equal to ten, and **b** is of length ten. The expression would return those elements of the first ten elements of **a** for which the corresponding element of **b** is greater than three. The subsetting can be continued with more sets of square brackets. Each set will operate on the result of all previous subsetting.

Matrices

For matrices, both kinds of square brackets are also used. For single square brackets, the selection instructions for the rows are separated from the selection instructions for the columns by a comma. Or, by not using the comma, the matrix is treated like a vector, going down the rows starting

with the first column. Like the subsetting for vectors, for single square brackets, indices or a logical expression may be used to subset a matrix. To reference all rows of a matrix, put nothing to the left of the comma inside the brackets. To reference all columns of a matrix, put nothing to the right of the comma inside the brackets.

Double square brackets return just one value. If subsetted with a row and a column index separated by a comma, the value in the cell is returned. If just one index value is entered within double square brackets, R treats the matrix as a vector—going down rows—and returns the indexed element of the vector.

An example of matrix subscripting is

```
a[ a[,1]>3 , 1:4 ],
```

where **a** is a matrix with at least four columns. The expression would return those rows of the first four columns for which the elements of the first column are bigger than three. Notice that the **a[,1]** consists of one column and contains all of the rows.

A matrix can also be subsetted using a matrix with two columns. The two-column matrix would contain row and column indices and would pick out individual cells in the matrix based on the indices in each row. For example, if **b** is a matrix with [1 2] in the first row and [2 3] in the second row, then a[b] would return the two elements a[1,2] and a[2,3].

Arrays

Arrays are like matrices but can have more than two dimensions. Note that a matrix is an array with two dimensions and a vector is an array with one dimension. Subscripting arrays with more than two dimensions is just like subscripting matrices except that, for single square brackets, there are more commas in the brackets. An example is

```
a[ 1:3,,2:7 ],
```

where **a** is a three-dimensional array with at least three levels in the first dimension and at least seven levels in the third dimension. The result of the subsetting would be all of the elements in the second dimension for which the index in the first dimension is one, two, or three and the indices in the third dimension are between two and seven inclusive.

Like matrices, arrays can be subsetted using a matrix that has the same number of columns as the number of dimensions of the array, the rows of which would consist of indices for individual cells of the array. Single square brackets with no comma and double square brackets work the same as with vectors and matrices.

Lists

Lists are collections of R objects (and a kind of vector). The objects can be any type of object and do not have to be of the same type within a list. The objects are indexed in the list. To look at objects in a list, single square brackets are used. For example,

```
blist[ 1:5 ]
```

would return the first five objects in **blist** and would also be a list.

To access an object in a list, double square brackets or a dollar sign are required. For example,

```
blist[[ 2 ]]
```

would return the second object in the list **blist** and

```
blist$b1
```

would return the object in **blist** with name **b1**. Objects in a list can only be accessed one at a time.

If a list is created from objects that do not have names associated with the objects, names will be given to the objects when the list is created. The names can be changed at any time.

Data frames are a special kind of list. Data frames have the same number of elements for every object in the list and are defined as a `data.frame`. Each object in the list is of one atomic mode (to be described in Chapter 4), though the different objects need not be of the same mode. Data frames can be subsetted like a matrix or like a list. If subsetted like a matrix, the resulting object will be a list. If subsetted like a list, the resulting object will be raw, complex, numeric, logical, or character depending on whether the list object is raw, logical, numeric, complex, or character. Individual cells in a `data.frame` can be accessed using indices in the double square brackets. For example,

```
adframe[[ 1,2 ]]
```

would return the element in the first row and second column of the data frame **adframe**.

Many functions return output in lists. Dollar sign subscripting is usually used to access the output, although square bracket indexing can be used. For example, for the linear model function `lm()`, entering

```
lm( y~x )$resid
```

or

```
lm( y~x )[[2]]
```

will return the residuals from a simple linear regression of y on x, as will the two sets of statements

```
a=lm( y~x )
a$resid
```

or

```
a=lm( y~x )
a[[2]]
```

Other Types

Other types of object can be subsetted, for example, factors and slots. Objects that are factors are vectors and can be subsetted like vectors. Slots are S4 objects and are subsetted using **@**. Slots should never be subsetted except in a method statement, which will be described in the chapter on functions. More information about subsetting both can be found by entering **??"Extract or Replace"** at the R prompt or by using the R Studio Help tab.

Odds and Ends

Two main object systems—S3 and S4—are used in R. Slots are part of S4. S3 and S4 are discussed throughout the book and in the pdf at www.r-project.org/conferences/useR-2004/Keynotes/Leisch.pdf.

Assignments can be done to subsets of an object. For example, let **a** be a matrix, and let the user want to change those values in **a** that are greater than 100 to 100. Then, the statement

```
a[ a>100 ] = 100
```

will do the replacement and leave the rest of the matrix intact.

In R Studio, the help pages are easy to access and have their own window. In R the **?** and **??** operators open the help pages. For known function names, ***?name*** (or **help(*name*)**) will return the help page for the function, where *name* is the name of the function. To search for functions related to some techniques or methods, the operator, ***??*** is used. Entering ***??"keywords"*** (or **help.search("*keywords*")**), where *keywords* consists of keywords about the technique or method, may give a list of functions in packages related to the topic. Sometimes, the search comes up blank. Try again with different keywords.

The colon is used in four ways in R. Of interest here is just the use of a single colon to define a sequence and the double colon to refer to functions by package and name.

If **a** and **b** are two numbers, the expression **a:b** will give the sequence of integers between **a** rounded down to an integer and **b** rounded down to an integer. Note that the number **a** can be larger than the number **b**.

The functions that come with R are all part of some package. If a package is not loaded, a search using just the function name will return nothing. The full name of a function is package.name::function.name, where *package.name* is the name of the package and *function name* is the name of the function.

For more information on colons, enter **?":"** at the R prompt or **:** under the Help tab in R Studio.

The operator ~ is used in model formulas to separate the left and right sides of a model. For more information, **type ?"~"** at the R prompt or enter ~ under the Help tab in R Studio.

The symbol # is used for comments. When writing functions, anything found to the right of a # on a line of the code is ignored.

The operator **%in%** returns TRUE for the values in the object to the left of the operator that are in the object to the right of the operator and FALSE for those that are not. The length of the result is the length of the first object. If the first object has more than one dimension, it is converted to a vector to get the result.

The CRAN help pages for subsetting are found by entering **??"Extract or Replace"** or by using the R Studio Help tab.