Chapter 3

Object Types in R

Having seen the kinds of things R can do, I'm sure everyone is anxious to get right on to revolutionary statistical discoveries and the production of eye-popping, full-color graphics. Before going there, we've got to stop for a moment and talk objects. This, I'm afraid, may be unpleasant. But objects are at the center of the R worldview, and misunderstanding them is the central cause of frustration for new, and even not so new, R users.

For better or worse, R is an object-oriented language. Being object oriented means that R procedures recognize the kind of object they are being called to process and behave differently depending on the object type. This is a great thing in that one command can adapt to many different situations. The **`summary()`** command, for example, recognizes whether you are trying to summarize a data set or the results of a regression analysis and adjusts to the differences between these two kinds of output. This can also be the source of some frustration in that the same command will do different things depending on the nature of your object. While R always has very strong views about what your object is, you will sometimes be confused on this point. This can lead to some unpleasantness in as much as R isn't really interested in your opinions on the subject.

When something isn't working the way you expect, object type is one of the first things you should check. In this chapter, we'll go over a taxonomy of object types and their basic care and feeding. I have to warn you up front that I am going to propose an approach to thinking about R

©SAGE Publications

objects that is mostly straightforward and coherent and, therefore, is not precisely consistent with R's own internal logic and terminology.[1]

There is a lot of technical detail surrounding objects, much of which I hope we will be able to avoid here. Part of the confusion about R objects comes from the fact that the term *objects* is really used for several different things that have different dimensions, characteristics, and purposes. Some of these concepts are technical legacies from the historical development of R. Others stem from the internal needs of the R program but are relatively remote from the experiences of everyday users. I'll point out a few instances where you may observe some inconsistency between my approach and R's more official notions; but I believe that the object schema I am going to propose will help you work through most of the object issues you are likely to encounter up through a relatively advanced level.

We'll start with a little housekeeping on naming and managing objects generally, and then I'll set out my approach to understanding R objects.

R OBJECTS AND THEIR NAMES

Objects in R can be thought of as simply the things that the program is keeping track of for you. These can be individual values, data sets, statistical outputs, or specialized functions. If it is something to which you can assign a name, it is an object.

Naming objects in R is subject to just a few rules. R names can be any combination of letters and numbers. No special characters (&, ^, %, $, #, @, etc.) are allowed except for periods and underscores. R names cannot start with a number; that will just confuse R. (This may give you some brief satisfaction in the sense that turnabout is fair play, but ultimately, confusing R always hurts you more than it.)

No spaces are allowed in R names. If you want to string words together for your variable name, the two most common practices are to demarcate the words with either periods or "camel capitalization." If different variables are for different years, for example, you could use a series of names like **`result.2005, result.2006, result.2007`**. Another common convention is to use an extension after a period to help keep track of the kind of object: **`result.df`** for a data frame, **`result.f`** for a function, and so

1. If you want the official view, use **`help(mode)`**, **`help(type)`**, **`help(class)`**, and **`help(method)`**. Good luck!

©SAGE Publications

on. You can use as many periods as you want for clarity, but more than two is probably going to start looking silly. If the variable names in an imported data set have spaces, R will helpfully replace them with periods (see Chapter 4 on importing data).

Camel capitalization is the practice of capitalizing the first letter of different strung-together words: NormalizedValue, FirstCut, AnnualRate. Just remember R's sensitivities around the issues of case. MyVariable has to be always MyVariable. R will feign ignorance if you ask it to do something with myVariable, Myvariable, or myvariable.

That's the deal on naming. Now we need to think about the things that you can give names to. Let's start with the simple dichotomy of data objects and nondata objects. The nondata objects shouldn't cause you too much grief. There are just two kinds that you are likely to encounter on a regular basis: functions and the output from statistical procedures. I'll go over customized functions in Chapter 7. The output from a statistical procedure is really just a list, one of the data types we'll look at in a moment. Everything we are going to learn about working with lists can be applied to working with statistical output. The most interesting things to do with statistical output involve treating it as a data source in itself, so there isn't too much else that needs to be said about that at this point. I do demonstrate a number of statistical procedures and discuss working with their output in Appendix B, if you are anxious for that.

The real place where object-type problems are going to arise is in working with data objects. R's approach to classifying data objects is, in my view, hopelessly confusing for new users. I am going to propose an alternative schema that largely parallels the official R approach but slightly shifts some of the concepts and terminology. There are, of course, some dangers in doing this. I'll try to point out some of the places where you may observe differences between official R and my approach. In practice, though, I don't believe that you will encounter operational problems with the approach I lay out here until you reach a very high level of R programming. By then, you will be more than ready to deal with the technical details on their own terms.

HOW TO THINK ABOUT DATA OBJECTS IN R

I propose that we think about R data objects as containers that hold data. There are, in this schema, just four types of containers we need to focus on. Each of these types of containers has three critical characteristics. The

©SAGE Publications

first characteristic is which of the four types of container it is. We'll label this the "object type," recognizing that officially R uses the term *object type* a little more broadly. The four object types we'll use are vectors, matrices, data frames, and lists.

The second characteristic of each object type is a description of the kinds of things that the container is holding. We'll call this characteristic the object "storage mode." This is a term that R uses as well, but we'll apply it slightly differently. For us, the storage mode is a description of the kinds of things that can be held in one of our data containers. There are three main storage modes in this approach: logical, numeric, and character. The numeric category can be further divided into the integer and the double mode. I am also going to identify two of what I call "pseudo storage modes": dates and factors.

Here is where the R overlords are beginning to squirm. They know that in official R, a list is really a storage mode. But I find it more sensible to think of lists as data containers. As we'll see, lists hold all kinds of data (and other things) and, thus, are more like containers than like data elements. Until you get into pretty high-level programming, this alternative conceptualization won't cause any problems. Meanwhile, while you will find much about dates and factors in the R literature, you won't find anything about pseudo storage modes. I just made that up. If you will indulge me for the time being, I will make the case for that characterization after we have covered the more conventional parts of my typology.

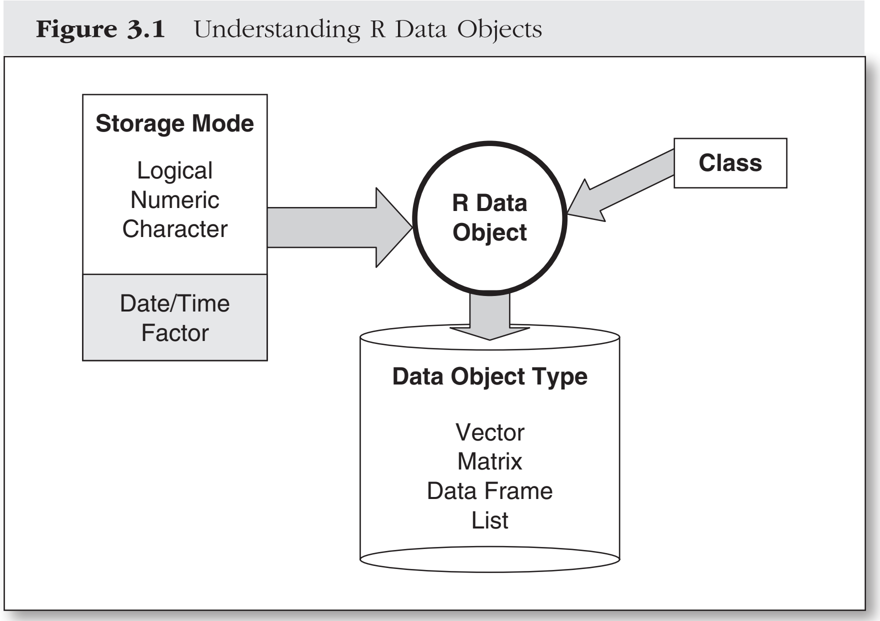
The third descriptor for our R objects is the object class. The object class is a label that provides more specialized information on how R should work with an object. While the number of object types and storage modes is quite limited, there are a wide variety of classes. The good news is that they rarely cause problems, so we can deal with them more generically.

Every R data object type, our data containers, can be described by what kind of container it is, what kind of data elements it contains, and the class descriptor that provides more specific information about how it should be treated. This approach to understanding R objects is illustrated in Figure 3.1 and summarized in Table 3.1.

All this would be pretty straightforward, except for the two pseudo storage modes that you can see lurking under the gray shading on Figure 3.1 and Table 3.1. We would prefer not to talk about these in polite company, but they are so critical and so widely used that we cannot exclude them from our discussion. Still, let's hold off on that for a bit. As is customary, let's start by talking about the well-behaved kids.

©SAGE Publications

Figure 3.1 Understanding R Data Objects



R Object Storage Modes

I propose thinking of object modes as the way R has the computer store the individual pieces of data that make up an object. For our purposes there will be three of these: logical, numeric, and character. The numeric storage mode can be further divided into integer and double values. I give a short description of each of the storage modes in Table 3.2.

The commands **`typeof()`** and **`mode()`** will usually tell you the storage mode, again with the caveat that R officially considers "list" a storage mode as well. The **`mode()`** command lumps together integer and double under the "numeric" label, while **`typeof()`** makes those distinct.

```
> # Data objects
> myInteger = as.integer(4)           # An integer (whole number)
> myWholeNumber = 5                   # A whole number stored as double
> myDouble = 3.7                      # A numeric-double number
> myOtherInteger = as.integer(3.7)    # Non-whole num convert to integer
```

©SAGE Publications

```
> myLogical1 = TRUE                 # A logical value set to TRUE
> myLogical2 = FALSE                # A logical value set to FALSE
> myCharacter = "Hello World!"      # A character string
```

Table 3.1 A Typology of R Objects

			Description
Nondata objects		Functions	Sets of commands bundled together in a custom function
		Output	The results of a statistical procedure
Data objects	Storage mode	Logical	TRUE/FALSE values
		Numeric (integer or double)	The integer mode holds whole numbers. The double mode can hold any number.
		Character	Textual data
		Date/POSIX	Numbers that can be interpreted as dates or times
		Factor	Categorical variables
	Data object type	Vector	A set of data elements all of the same storage mode
		Matrix	A two-dimensional array of data elements all of the same storage mode
		Data frame	A set of vectors in which the vectors (columns) do not all have to be of the same storage mode
		List	A collection of other objects
	Data object class		An indicator that tells R what specialized methods to use on an object

©SAGE Publications

Table 3.2 R Storage Modes

Storage Mode	Subcategory	Description
Logical		TRUE/FALSE values
Numeric	Integer	Numbers without fractional parts
	Double	Numbers that may have fractional parts
Character		Text values e.g., "Horse", "Alligator", "Four score and seven"
Date	Date	Calendar dates
	POSIX	Dates and times
Factor	Unordered	Categorical values without a clear ordering e.g., "Dog", "Cat", "Wombat"
	Ordered	Categorical values with an order e.g., "short", "tall", "grand", "venti"

```
> typeof(myInteger)                       # Test "typeof" for integer
[1] "integer"

> mode(myInteger)                         # "mode" for integer
[1] "numeric"

> typeof(myWholeNumber)                   # "typeof" for whole number
[1] "double"

> typeof(myDouble)                        # "typeof" for double
[1] "double"

> mode(myDouble)                          # "mode" for double
[1] "numeric"
```

©SAGE Publications

```
> myOtherInteger                      # Show 3.7 converted to integer
[1] 3

> typeof(myOtherInteger)              # "typeof" for convert to integer
[1] "integer"

> mode(myOtherInteger)                # "mode" for convert to integer
[1] "numeric"

> typeof(myLogical1)                  # "typeof" for logical
[1] "logical"

> typeof(myCharacter)                 # "typeof" for character
[1] "character"
```

Again, I've left the two pseudo storage modes in the gray shading for later. Officially, there are also a few other R object storage modes that you don't want to encounter down some dark programming alley the night before a statistics assignment is due: Complex, NULL, Raw, Closure, Special, Builtin, and Environment, for example, are either for special cases or for R's own internal purposes. For now, let's look more closely at the character, numeric, and logical storage modes.

The Character Storage Mode

The character storage mode is used for strings of text. You will see these referred to by all of those terms: "character," "text," or "string." You can usually tell that a value is in character mode when it is surrounded by quotation marks. You can check for character mode with **`mode()`** or **`typeof()`**, or with **`is.character()`**. There are two things of note to be careful about here. First, the backslash is R's escape character, so you can't include it in a text string without some extra steps.[2] You'll have problems, then, if you are trying to import data that include backslashes. This, happily,

2. I will go over this and much more about working with text data in Chapter 8.

©SAGE Publications

is a relatively rare situation. The second, and more common, issue is that R can store numbers in character mode. This is highly vexing in that it often happens without your knowing it and you cannot perform numeric operations on character data. You can tell when a number has been stored as a character because it will be displayed within quotation marks. As we'll see in a moment, this happens because all the objects in a vector or a matrix have to be of the same storage mode. If you try to create a vector or a matrix with a set of values that has character data in it, it will convert the whole set to the character storage mode. We'll come back to the problem of number/character confusion and what to do about it after we've confronted factors, which complicate the story even further.

The Numeric Storage Modes

The numeric storage modes are pretty straightforward. On occasion, you'll have to be a little careful about the distinction between integer and double mode. An integer, of course, is a whole number, a number that doesn't have a fractional part (1, −3, 7, 2,568,482, as opposed to 1.65, 2.9, −4.002). In R, integers can be stored either in integer or in double mode.[3] This can be a little confusing since, for example, the command **`as.integer()`** truncates the fractional component of a number to turn it into and store it in the integer mode. But **`is.integer()`** is a test of mode that returns **`FALSE`** unless the number is, in fact, stored in R's integer mode. If you need to test for whether something is an integer, in the sense of not having a fractional part, use **`is.wholenumber()`**. The good news is that since both double and integer are numeric modes, this difference shouldn't cause problems.

The Logical Storage Mode

The logical storage mode registers whether something is true or false. Logical values look a little like character values but act a little like numeric values. They look like character values in that R displays logical values as either TRUE or FALSE. In accord with their existential importance, TRUE and FALSE are always capitalized in R. Likewise, note that they do not use

3. The "double" label comes about because most programming languages use a distinction between "single" and "double" precision for how much space is dedicated to storing a number.

©SAGE Publications

quotation marks (which is how you can tell that they are not really in character mode). You can shorten logical values to T or F (again with no quotation marks). Just be careful that you don't ever name any variables T or F, in which case, things will get seriously confused.

The logical storage mode acts like numeric values in that for some operations R will treat them as zeros and ones. You can, for example, add or multiply logical values. You can use them in **`sum()`** or **`mean()`**, or in a regression model. In all these cases, they act just like numbers, with TRUE = 1 and FALSE = 0.

Once you have a group of character, numeric, or logical values, you will want to keep them in one of our data containers: the R data object types.

R DATA OBJECT TYPES

In my approach, the basic object types are containers that can hold some collection of data. For this purpose, I propose focusing on four basic object types: vectors, matrices, data frames, and lists. These object types hold particular data structures, as shown in Table 3.3.

Table 3.3 Basic Data Object Types in R

Object Type	**Description**
Vector	A one-dimensional set of values All values have to be of the same storage mode
Matrix	A two-dimensional array A matrix will be indexed by rows and columns. All elements have to be of the same storage mode
Data frame	A data set organized with variables in the columns and observations in the rows May hold objects with different storage modes, although each variable (column) has to be of the same storage mode
List	A collection of other objects Can mix all other types and storage modes

©SAGE Publications

Again, my approach is a little different from that of official R. For official R, a list is a storage mode. If you enter **mode(myList)**, the answer you will get is "list." A data frame is stored as a list. If you enter **mode(myDataFrame)**, the answer will also be "list." For our purposes, a list is more like a data container than a storage mode. It is something in which you keep data. Meanwhile, the data frame is such a special form of list that it is worth treating it independently as one of our object types.

There is no regular R function that quickly tells you the object type in the sense that I use here. There are, however, individual tests you can use to figure each of these things out: **is.vector()**, **is.matrix()**, **is.data.frame()**, and **is.list()**. Jumping ahead a bit, we'll look at a custom function in Chapter 7 that is a simple procedure to join these individual tests together to indicate the object type. I think it is sufficiently straightforward to reproduce here, even though we are still some distance from doing custom functions.

```
DOType = function(x){                # DOType function ------------------+
# This is a function to identify data object types. I think             |
# of object type as a characterization of objects that hold collections |
# of things. These object types are vectors, matrices, data frames,     |
# and lists.                                                            |
# If none of those types fit, then the function returns a statement     |
# that it is not a recognized data type.                                |
#                                                                       |
  DOT = ""                           # Set default value for DOT        |
  if(is.vector(x)){DOT = "vector"}   # Check if is a vector             |
  if(is.matrix(x)){DOT = "matrix"}   # Check if is a matrix             |
  if(is.data.frame(x)){              # Check if is a data frame         |
    DOT = "data frame"}              #                                  |
  if(is.list(x) & !is.data.frame(x)){ # Check if is a list (and not a   |
    DOT = "list"}                    #   dataframe)                     |
                                     #                                  |
  if(DOT == ""){DOT = paste("Not a", # Print a message if it is none    |
    "recognized data object type")}  #   of the above                   |
  return(DOT)                        # Return the appropriate value     |
}                                    # End of function ------------------+
```

©SAGE Publications

Once set up, the **DOType()** function can be used to indicate data object types. Now, let's look a little bit more into the care and feeding of each of these data object types.

THE BASIC DATA OBJECTS: VECTORS

When we are thinking about data structures in R, it is particularly important to understand the basic concepts of vectors and matrices. R is a vector-based language. This is the source of much of its power, but it also contributes to making R confusing for new users.

A vector is simply a grouped set of values. It is like a list, except that I won't call it a "list" because, as we've seen, that is another specific object type in R. When you import data from an external source, R usually brings it in as a set of vectors: one for each variable. The power of vector-based operations in R comes from the fact that when you specify an action to take with a vector, R applies that action to every element in the vector. A single command operates on the whole group of elements.

For our purposes, vectors can be made up of elements from any of the four object storage modes: logical values, numbers (integers or double precision), or characters. As I have emphasized, all of the elements in any one vector have to be of the same storage mode. In R's view, the vector itself has a mode that it inherits from the data it contains.

The concatenation function, **c()**, is used to package a set of things up into a vector. **myVector = c(1, 2, 7, 9)** tells R to create a vector with the numbers 1, 2, 7, and 9 and to assign it to an object named **myVector**.

```
> myLogicalVector = c(T, F, T, T)       # Set up a logical vector
> myNumericVector = c(1, 2, 4, 7)       # Set up an integer vector
> myTextVector = c("a", "b", "7", "x")  # Set up a text vector
> typeof(myLogicalVector)               # Show type for logical vector
[1] "logical"

> typeof(myNumericVector)               # Show type for numeric vector
[1] "double"

> mode(myTextVector)                    # Show type for text vector
[1] "character"
```

©SAGE Publications

The special case of a vector with only one element is called a scalar. You can think of scalars as constants. Since scalars have only a single element, they don't need to be concatenated and can be created without using the **c()** function.

```
> myScalar = 1                       # Create a scalar with value 1
> yourScalar = 2                     # Create a scalar with value 2
> myScalar + yourScalar              # Add the two scalars
[1] 3
```

Because vectors must contain data with the same storage mode, if you create a vector **my.vector = c(1, 2, 7, "Smith")**, R will have to do something to make those elements consistent. R's approach to this is to assume that 1, 2, and 7 are characters rather than numbers, because "Smith" is clearly a character element. If you create such a vector and then try to add the first two elements, you will get an error that this is nonnumeric data.

There is a little more fluidity between logical and numerical data. If you try to use logical data in a numerical context, they will be treated as 1s and 0s. Likewise, if you try to use numerical data in a logical setting, they will be interpreted as TRUE wherever the numerical value is not zero and as FALSE wherever it is zero. While you'll get an error trying to mix operations with character and numeric vectors, logical vectors can be used in operations with numeric vectors. The former may be more frustrating, but the latter can be more dangerous, since you can think that something is working properly when it is not.

```
> myVector1 = c(0, 5, 18)              # Set up a numeric vector
> typeof(myVector1)                    # Check vector type
[1] "double"

> myVector2 = c(TRUE, TRUE, FALSE)     # Set up a logical vector
> typeof(myVector2)                    # Check vector type
[1] "logical"

> myVector3 = c("Fred", "Joe", "Simon") # Set up a character vector
> typeof(myVector3)                    # Check vector type
[1] "character"
```

©SAGE Publications

```
> myVector1 + myVector3                # Add num & char vectors (error)
Error in myVector1 + myVector3: non-numeric argument to binary operator

> myVector1 + myVector2                # Add numeric & logical vectors
[1]  1  6  18

> as.logical(myVector1)                # Treat numeric as logical
[1] FALSE  TRUE  TRUE
```

In this example, we see the different data types and how logical values are treated as 0 or 1.

Vector Indices

The particular elements within a vector are identified with an index number enclosed in brackets at the end of the vector name. If **myVector = c(3, 9, 5)**, then **myVector[2]** is the second element, 9, and **myVector[3]** is the third element, 5.

```
> myVector = c(3, 9, 5)                # Set up vector of numeric values
> myVector[2]                          # Get the vector's 2nd element
[1] 9

> myVector[3]                          # Get the vector's 3rd element
[1] 5
```

You can also use another object to identify the element in the vector, as in the following example:

```
> myVector = c(3, 9, 5)                # Set up vector of numeric values
> myIndex = 1                          # Set up a selection index
> myVector[myIndex]                    # Use index for selection
[1] 3
```

©SAGE Publications

Here is another example with a vector of text elements and a selection vector:

```
> myVector = c("Bob", "Mary", "Fred")   # Set vector of character values
> myVector[2]                           # Show 2nd element
[1] "Mary"

> myIndex = c(1, 3)                     # Set up an index variable
> myVector[myIndex]                     # Select vector elements w/index
[1] "Bob"  "Fred"
```

This method of element selection is very powerful because it allows us to choose vector elements conditionally. We'll get to that in much more depth in Chapter 6.

Vector Operations

When vectors are used in operations, there are four main ways in which the operation can work. Most of the time, these modalities are reasonably intuitive. Every once in a while, they'll jump up and bite you when you aren't being careful.

Vectorized functions work on each element of the vector and produce a new vector with the same number of elements. For example, **`log()`** creates a new vector with the log of each element from the input vector.

```
> myVector1 = c(1, 2, 3)                # Create vector with 3 elements
> myVector2 = log(myVector1)            # Create new vector with log
> myVector2                             # Print new vector
[1] 0.0000000 0.6931472 1.0986123
```

Nonvectorized functions can only work on the individual elements of the vector. In this case, you have to be explicit about which element in the vector you want R to use. For example, **`if()`** requires a single element, while **`ifelse()`** is vectorized.

©SAGE Publications

```
> myVector = c(1, 2, 3)                # Numeric vector with 3 elements
> if(myVector == 1) print("Answer is 1")# This produces a likely error
[1] "Answer is 1"
Warning message:
In if (myVector == 1) print("Answer is 1") :
  the condition has length > 1 and only the first element will be used

> if(myVector[1] == 1)                 # This time specify 1st element
+   print("Answer is 1")
[1] "Answer is 1"

> myAnswer = ifelse(myVector == 1,     # Here is the vectorized ifelse()
+   "Answer is 1",                     #   it operates on each vector
+   "Answer is not 1")                 #   element individually
> myAnswer                             # Show results
[1] "Answer is 1"     "Answer is not 1" "Answer is not 1"
```

The difference between vectorized and nonvectorized functions is critical. Failure to understand which way a function works will lead to much grief. In Chapter 7, we'll look at the **apply()** family of functions, which can help nonvectorized functions operate on vectors.

A third kind of operation works on the whole vector: **min()**and **max()**, for example, return a single value that is the minimum or maximum element from the vector; **sum()** gives the sum of all the elements of the vector.

```
> myVector = c(1, 2, 3)                # Create a vector

> min(myVector)                        # Get the vector minimum
[1] 1

> max(myVector)                        # Get the vector maximum
[1] 3
```

©SAGE Publications

```
> sum(myVector)                       # Sum the vector elements
[1] 6
```

Finally, there are operations that work on multiple vectors. Addition, for example, adds the elements of one vector to the elements of another with pairwise matching: The first element of the first vector is added to the first element of the second vector, and so on. Reasonably enough, there need to be the same number of elements in each vector.[4] Multiplication and division work the same way: element by element.

```
> myVector1 = c(1, 2, 3)              # A vector with 3 elements

> myVector2 = c(10, 20, 30)           # Another 3-element vector

> myVector1 + myVector2               # Vector addition
[1] 11 22 33

> myVector1 * myVector2               # Vector multiplication
[1] 10 40 90

> myVector2/myVector1                 # Vector division
[1] 10 10 10

> myVector5 = c(10, 20, 30, 40, 50)   # A vector with 5 elements

> myVector1 + myVector5               # Add different-length vectors

Error in myVector1 + myVector3 : non-numeric argument to binary operator
> myVector6 = c(10, 20, 30, 40, 50, 60) # A vector with 6 elements
```

4. To be perfectly honest, it requires that either the same number of elements or the number of elements in the longer vector is a multiple of the number of elements in the shorter vector, in which case, addition keeps reusing the shorter vector. (Did you get that?)

©SAGE Publications

```
# Add different-length vectors where one length is a multiple of the other
> myVector1 + myVector6
[1] 11 22 33 41 52 63
```

In this example, we see element-by-element mathematical operations. Note the error when the two vectors are of different lengths. But different lengths are okay if one is a multiple of the other.

Vectors are unidimensional. They contain one set of data elements, all of the same type. Matrices add a second dimension. Unlike vectors, you actually won't use matrices all that much; as we'll see, a data frame is a much more useful container for two-dimensional data. Nonetheless, matrices are conceptually important, and there are some operations that only work with matrices.

THE BASIC DATA OBJECTS: MATRICES AND THEIR INDICES

A matrix has two dimensions, a row dimension and a column dimension. You can think of it as a single object that contains a set of row vectors or a set of column vectors. In fact, it contains both, overlapping. You will recall from our discussion in Chapter 1 that a data set is usually a two-dimensional structure with observations and variables. A data set, then, is essentially a matrix in which each variable is a column vector and each observation is a row vector (see Table 3.4).

Table 3.4 A Data Matrix

	Variable 1	**Variable 2**	**Variable 3**
Observation 1	15	3.5	2
Observation 2	10	2.1	5.8
Observation 3	8	5	7

©SAGE Publications

Anyone who has played the game Battleship understands the indexing for the elements of a matrix. A matrix is indexed by two values: a row index and a column index: **myMatrix[row,column]**. By convention, the row index always comes first. This is important. You've got to commit this to memory right now: *row first, then column*. **myMatrix[3,19]** is the element in the 19th column of the 3rd row of the matrix **myMatrix**.

We can isolate a specific row or column vector by leaving one of the two indices blank:

myMatrix[,4] references the 4th column of **myMatrix.**

myMatrix[12,] references the 12th row of **myMatrix.**

Thinking in terms of a data set, we could say that **myMatrix[,4]** is a list of the values for the fourth *variable* for every observation in the data set. **myMatrix[12,]** is a list of all of the variable values for the 12th *observation* in the data set.

As with vectors, basic mathematical operations work on the corresponding individual matrix elements. **myMatrix + 7** adds 7 to *every* element in **myMatrix**. **myMatrix1/myMatrix2** divides each element in **myMatrix1** by the corresponding (i.e., same row and column) element in **myMatrix2**. R's normal matrix operations are all by corresponding element, which means, as with vectors, that the two objects have to be either of the same dimensions or a multiple thereof. Those of you looking for authentic matrix algebra, as is likely in a more advanced statistics class, need to enclose the operator in percent signs (see Chapter 7 for more discussion on mathematical operators).

```
> myMatrix = rbind(c(3, 8), c(23, 33))     # Create matrix by binding 2 rows
> myMatrix                                 # Display myMatrix
       [,1] [,2]
[1,]     3    8
[2,]    23   33

> myMatrix1 = myMatrix + 7                 # Add 7 to each myMatrix element
> myMatrix1                                # Display myMatrix1
```

©SAGE Publications

```
      [,1] [,2]
[1,]   10   15
[2,]   30   40

> myMatrix2 = rbind(c(2, 5), c(3, 2))   # Create another matrix
> myMatrix2                             # Display myMatrix2
      [,1] [,2]
[1,]    2    5
[2,]    3    2

> myMatrix3 = myMatrix1/myMatrix2       # Divide myMatrix1 by myMatrix2
> myMatrix3                             # Display myMatrix3
      [,1] [,2]
[1,]    5    3
[2,]   10   20

> myMatrix4 = myMatrix2 %*% myMatrix3   # Matrix algebra multiplication
> myMatrix2                             # Display myMatrix2
      [,1] [,2]
[1,]    2    5
[2,]    3    2

> myMatrix3                             # Display myMatrix3
      [,1] [,2]
[1,]    5    3
[2,]   10   20

> myMatrix4                             # Display myMatrix4
      [,1] [,2]
[1,]   60  106
[2,]   35   49
```

Mathematical operations with matrices work on a corresponding element-by-element basis. This is different from real matrix algebra. Be careful!

R matrices, like R vectors, can only be made up of the same kind of data, for example, character, logical, or numeric data. So R matrices don't work for data sets that include mixed data. For that, we need to turn to data

©SAGE Publications

frames, which are a more flexible matrix-like structure that can hold variables with different storage modes.

The Basic Data Objects: Data Frames

Data frames are the central structures for holding and organizing data in R. Data frames look like matrices. They use the same **`[row,column]`** indexing but are more useful both because they can hold variables with different storage modes and because they allow referencing of variables by name. Note that while the data frame can hold variables of different storage modes, all of the elements of a given variable (column) still have to be of the same mode.

The ability to hold data with different storage modes is important, but it is the referencing by name that is the real key to the power and utility of data frames. In the following example, we create a data frame with the **`data.frame()`** command. We can then access the variables with several kinds of notation.

```
myData = read.table("mydata", header = TRUE)
myDataframe = data.frame(myData)

myDataframe$variablename
myDataframe["variablename"]
myDataframe[,"variablename"]
myDataframe[,column number]
myDataframe[column number]
```

All five of these approaches do the same thing. The first is the clearest and the most commonly used.

The second and last approaches show how the columns (variables) are privileged in a data frame: If there is only one index, it is assumed to refer to the columns rather than the rows. If you try all these ways, you'll see that the printing behavior changes. When you use the variable names, R prints the data in a table with the row names as well as the column names. When you use the numeric identifiers, it just prints the vector of values.

©SAGE Publications

R assigns both variable names and observation names to the data frame. The variable names can be drawn from headers if you are reading in a table of data from an external source.

Unless you change them, the row names will just be observation numbers. You can give the observations names to help identify them, for example, the names of states, experiment numbers, or names of students. To set the row names, you can use the **rownames(myDataframe)** command. The same approach also works for column names: **colnames (myDataframe)**. In this example, I show how to set column names independently and to set them to be the same as the values in the first row of the data frame.

```
rownames(mydata) =                    # Add row names by hand
   c("Mary", "Michael", "Mia", "Michelle", "Mark")

rownames(mydata) = mydata[,3]         # Row names = third col. in data frame
colnames(mydata) = c("sex", "age")    # Add column names by hand
colnames(mydata) = mydata[1,]         # Set col names to values in 1st row
                                      # Note that this keeps 1st row as
                                      #   1st obs in the data frame
```

Referencing Data Frame Elements

The use of the **myDataframe$myVariable** convention can get cumbersome. It puts a real premium on short data frame and variable names. The most common alternative is to "attach" the data frame using the **attach()** command. This tells R to look for objects in the attached data frame, so the data frame name is no longer required.

```
> myData.df = read.csv("mydatafile.csv", header = TRUE)

> attach(myData.df)
```

Note that no quotation marks are required for **mydata.df** *because it is an R object, while* **mydatafile.csv** *is an external object and so is referenced as a text name within quotation marks.*

©SAGE Publications

If you have used the **attach()**method and are having regrets, or are simply ready to move on to another data frame, you can detach the data frame with the **detach()**method.

```
> detach(myData.df)
```

attach() is a popular and convenient approach, and is therefore frowned upon by some of the R cognoscenti. To be fair, there are two problems with the **attach()** method. In the first place, **attach()** copies the variables from the data frame into the R workspace. Any changes you then make to the variables only affect the copies in the workspace, not the originals in the data frame. Second, if you are using multiple data sets with any overlapping object names, it is easy to get confused about where the objects are coming from and make mistakes. You should appreciate these concerns since, if I may be snide, it is not always the case that the R cognoscenti are observably concerned about the confusion of new users.

My recommendation would be to go ahead and use **attach()** if you are certain that you will only be using one data frame that hasn't simply been built from other objects that are already in memory. Otherwise, there are two other ways beyond the **dataframe$vName** approach to let R know which data frame a variable is associated with, which are a little less cumbersome than the **dataframe$vName** construction but are only applicable in certain circumstances.

You can sometimes specify the appropriate source for a variable within the command itself using the **data=** option. The command to construct a linear model, **lm()**, for example, allows the **data=** option to indicate the data frame. The **plot()** command, on the other hand, does not.

For a larger section of work, for example, the complex instructions for a plot, you can use the **with(dataframe, {commands})** function. This allows you to group a set of commands that utilize the same data frame. As with the attach method, however, you have to be careful in that any transformations to the data are only applicable within the **with({})** statement.

The **with({})** approach is useful for reasonably contained operations where you can keep track of the fact that a larger set of commands

©SAGE Publications

is contained within the **with({})** command. I wouldn't recommend it for more extensive processes where you might lose track of the data source and the need for a closing brace and parenthesis. The **with({})** approach is also constrained by allowing only one object to be the output.

Note also the somewhat unusual use of the parentheses and braces in the **with({})** process. The commands have to be one per line and are grouped both within the braces and the larger set of parentheses.

Here are a few demonstrations of all four methods of referencing variables within a data frame:

```
> myDF = data.frame(                    # Create a data frame
+   myVar1 = c(seq(0, 100, by = 5)),    # Set up variable 1
+   myVar2 = c(0:20))                   # Set up variable 2

> # 1. the $ construction ---------------------------------------------------
> mean(myDF$myVar1)                     # $ Always works, minimizes errors
[1] 50

> # 2. The attach() method -------------------------------------------------
> attach(myDF)                          # Attach the data frame
>   mean(myVar2)                        # Do something with the data frame
[1] 10

>   myVar2 = myVar2 * 2                 # Here is a transformation
>   mean(myVar2)                        # Mean of transformed variable
[1] 20

> detach(myDF)                          # Detach the data frame
> mean(myDF$myVar2)                     # Transform lost outside of attach
[1] 10

> # 3. The data = method ---------------------------------------------------
> lm(myVar1 ~ myVar2, data = myDF)      # Run a linear model w/data frame

Call: lm(formula = myVar1 ~ myVar2, data = myDF)

Coefficients:
(Intercept)       myVar2
  1.551e-15    5.000e+00
```

©SAGE Publications

```
> # 4. The with() method -----------------------------------------------------
> with(myDF, {                              # with() to indicate data frame
+   myVar1 = myVar1/2                       # Transform myVar1
+   sd(myVar1)                              # Std dev. of transformed myVar1
+ })                                        # Note close brace & paren finish
[1] 15.51209

> sd(myDF$myVar1)                           # Transform lost outside with()

[1] 31.02418
```

This example shows the four different methods for referencing data frame elements. The **$** *construction is the safest and most reliable, but it can get unwieldy. The* **data=** *method only works in some commands and functions. Variable tranformations made within the* **attach()** *and* **with()** *methods are only local, they don't persist outside that immediate environment.*

All of these methods work. My own view is that your default method should be the **dataframe$variable** construction. It is a little more cumbersome, but it always makes it clear what variable you are using and where it has come from. The other approaches should only be used where the gains in efficiency don't come at too high a price in terms of this clarity.

Displaying the Contents of a Data Frame

You can display the names of the variables in your data frame with the **names()** command. The whole data frame is displayed simply by typing its name. This isn't so helpful for big data sets. If you are using RStudio, just click on the name of the data frame in the workspace window (top-right quadrant) to see the data frame in a spreadsheet format. You can accomplish the same thing with the **View(myDataframe)** command (note the capitalization of **View()**). Summary statistics are available through the **summary()** command. In Chapter 5, we'll go over some other ways of viewing and reviewing larger data sets.

©SAGE Publications

```
> myVar1 = c("a", "b", "c")             # Create char variable
> myVar2 = c(10, 11, 12)                # Create numeric variable
> myDF = data.frame(myVar1, myVar2)     # Combine into data frame
> myDF                                  # Print data frame
  myVar1 myVar2
1      a     10
2      b     11
3      c     12

> names(myDF)                           # Show names of variables in myDF
[1] "myVar1" "myVar2"

> summary(myDF)                         # Summarize vars in data frame
    myVar1                 myVar2
 Length:3           Min.   :10.0
 Class :character   1st Qu.:10.5
 Mode  :character   Median :11.0
                    Mean   :11.0
                    3rd Qu.:11.5
                    Max.   :12.0
```

The **summary()***command gives the basic descriptive statistics for the numeric variable, but it cannot do much with the character variable.*

THE BASIC DATA OBJECTS: LISTS

Lists are perhaps both the most and the least important object types in R. They are the most important because lots and lots of things are packaged as lists. They are the least important because you don't often need to work with them directly. Still, for the former reasons, we have to spend a little time on them. Moreover, understanding a bit about lists will help you avoid some other potentially painful object lessons.

Lists are groups of objects. The important thing about lists is that they are pretty free form. A list can package together objects of different types

©SAGE Publications

(vectors, data frames, other lists, etc.) and of different storage modes (character, numeric, etc.). And all of those things can be of different lengths. For example, when R runs a linear model using the **lm()** command, it creates a list with the output. That list will include short elements with character data, such as the model used in the function call, and long elements with numeric data, such as a vector of residuals.

You can't find out what is in a list just by entering its name. R, in its infinite wisdom, decides what kind of list you've got (based on the class attribute, if it is included in the list) and then prints it out accordingly. If it is a data frame, it just prints the data. If it is a model, it prints the most relevant model output. To see what really lurks within a list, you need to use the **attributes()** command, which we'll look at in more detail shortly. **attributes(myList)** will display all the elements within the list, which you can then access by either using the **myList$element** construction or indicating the slot number of the item in the list: **myList[3]**. A linear model, for example, generates a set of residuals that are included in the second slot of a list that holds the output from the model. That element can be accessed with either **myModel$residuals** or **myModel[2]**.

```
> myVar1 = c(1:8)                        # Set up a y variable
> myVar2 = c(3, 5, 4, 6, 7, 9, 2, 9)     # Set up an x variable
> myModel = lm(myVar2 ~ myVar1)          # Create a linear model
> myModel                                # Print the model output

Call:
lm(formula = myVar2 ~ myVar1)

Coefficients:
(Intercept)       myVar1
     3.3214       0.5119

> attributes(myModel)                    # Show elements in model output
$names
[1] "coefficients" "residuals"  "effects"  "rank"  "fitted.values"
[6] "assign"  "qr"  "df.residual"  "xlevels"  "call"  "terms"  "model"
```

©SAGE Publications

```
$class
[1] "lm"

> myModel$residuals                    # Show model residuals
      1       2       3       4       5       6       7       8
-0.8333  0.6547 -0.8571  0.6309  1.1190  2.6071 -4.9047  1.5833
```

There is also a double-bracket approach to accessing the elements of the list, for example, **`myModel[[2]]`**. This is an important distinction because the double-bracket notation (along with the **`$`** demarcation) gives you access to the individual components within the list element, while the single-bracket approach only gives you access to everything in the list slot all at once. You can think of the single brackets showing you the box, while the double brackets give you access to the things inside the box.

The fact that data frames are lists usually shouldn't cause any problems, but if you are trying to automate something using the bracket method to get inside a data frame, you may need to recall the distinction between single- and double-bracket indexing.

```
> myModel[2]                  # Single-bracket index result
$residuals
      1       2       3       4       5       6       7       8
-0.8333  0.6547 -0.8571  0.6309  1.1190  2.6071 -4.9047  1.5833

> myModel[[2]]                # Double-bracket index result
      1       2       3       4       5       6       7       8
-0.8333  0.6547 -0.8571  0.6309  1.1190  2.6071 -4.9047  1.5833

> myModel[2][1]               # Single bracket won't open list item
$residuals
      1       2       3       4       5       6       7       8
-0.8333  0.6547 -0.8571  0.6309  1.1190  2.6071 -4.9047  1.5833
```

©SAGE Publications

```
> myModel[[2]][1]                      # Dble [[]] allow list item access
   1
-0.8333

> myModel$residuals[1]                 # $ referencing works the same way
    1
-0.8333
```

You can package things up in a list yourself using the **list()** command. Even lists can be included in other lists. The names of the slots in the list can be attached with the **names()** command. Then you can even build stacked descriptions to retrieve elements from specific slots. The following example puts the model output, itself a list, into another list with some additional information about the model:

```
> myList = list(1,                     # List starting with model num (1)
+   myModel,                           # Then the myModel list
+   "This is a discussion of myModel") # Then some discussion of myModel

> names(myList) = c("ModelNumber",     # Create the list names
+   "ModelOutput", "ModelDiscussion")
> myList                               # Show myList
$ModelNumber
[1] 1

$ModelOutput

Call: lm(formula = myVar2 ~ myVar1)

Coefficients:
(Intercept)       myVar1
     3.3214       0.5119
```

©SAGE Publications

```
$ModelDiscussion
[1] "This is a discussion of myModel"

> myList$ModelOutput$residuals          # Residuals from myModel in myList
   1        2       3       4       5       6       7       8
-0.8333  0.6547 -0.8571  0.6309  1.1190  2.6071 -4.9047  1.5833
```

This example shows how to bind together a new list that includes a preexisting list as one of its elements. We address an element in the list within a list in the final line with the two(!)-$ construction.

You can remove objects from a list with the **unlist()** function. But be forewarned that this will create a character vector with one entry for each of the things that were in the list. Except for the simplest of lists, you'll likely have to do some careful transformations to make anything useful out of it.

```
> myList = list(c(1, 2, 3), c("a", "b", "c"), "It's numbers and letters!")
> typeof(myList)                        # Show object type for myList
[1] "list"
> myList                                # Print myList
[[1]]
[1] 1 2 3
[[2]]
[1] "a" "b" "c"
[[3]]
[1] "It's numbers and letters!"

> myNotList = unlist(myList)            # New object = unlisted myList
> typeof(myNotList)                     # Show type of new unlisted object
[1] "character"
```

©SAGE Publications

```
>  myNotList                          # Show my new object
[1] "1"                         "2"                         "3"
[4] "a"                         "b"                         "c"
[7] "It's numbers and letters!"
```

In this example, we create and then disassemble a list.

A Few Things About Working With Objects

You can get a list of all of the objects in your current R session with either the **objects()** or the **ls()** command. Neither of these commands requires an argument, so you just leave the parentheses empty.[5]

```
> objects()                           # Show all active objects
[1] "myNumber"   "myRandNum"   "myVar"
```

One of the benefits of RStudio is that it shows a list of your current objects in the upper-right quadrant (see Figure 1.2). You can see what is in an object by clicking on it.

You can remove an object with the **rm(myObject)** function. You can remove all of your objects with **rm(list = ls())**. We'll learn about lists a little later. In the meantime, be a little careful with that one!

```
> myVector = 1:10                     # Create some objects
> myNewVector = myVector + 3
> myAnimal = "aardvark"
> objects()                           # List the objects
[1] "myAnimal"   "myNewVector"   "myVector"
```

5. Keeping track of your objects is another useful feature of the RStudio interface. Its object window lets you see the objects and display them in a spreadsheet-like format.

©SAGE Publications

```
> rm(myVector)                          # Remove an object
> objects()                             # List the objects
[1] "myAnimal"    "myNewVector"
> rm(list = ls())                       # Remove ALL objects
> objects()                             # List the objects
character(0)
```

*In this example, we tell R to remove myVector and then list the objects again. Note that "***`character(0)`***" is R's clumsy way of telling us that there are no objects left in the workspace.*

If you are using RStudio, you can also remove all objects with the Clear Workspace option under the Session menu or with the little broom button above the workspace window on the top right. In the plain R Console, there is a "remove all objects" option under the Misc menu.

It is important to remember that objects can be overwritten by new objects with the same name. This is a nice feature when you use it to keep redundant objects from piling up in your project. It isn't so nice if you lose track of your object values because you weren't aware that R was overwriting them. Here is a demonstration.

```
> myNumber = 5                          # Assign value 5 to myNumber
> myNumber = 7                          # Assign value 7 to myNumber
> myNumber                              # Show that 7 replaced 5
[1] 7
> myNumber = 5                          # Assign value 5 to myNumber
> myNumber = myNumber + 4               # Add 4 to myNumber
> myNumber                              # Show new value for myNumber
[1] 9
```

OBJECT ATTRIBUTES

R data objects are defined by a set of attributes. One of the most critical strategies when you get in trouble is to be sure you understand your

©SAGE Publications

objects. There are, inexplicably, two distinct commands for looking at the attributes of your objects: **attributes()** and **attr()**. They are very similar but work in slightly different ways. The **attributes()** function takes just one argument: the name of the object you want to look at. The **attr()** function takes two arguments: (1) the name of the object and (2) the name of the specific attribute. With the **attributes()** function, the specific attributes are appended with the **$name** approach, for example, **attributes(myObject)$class**.

The **attributes(myObject)** function is the most straightforward way to get a complete list of object attributes, while the **attr(object, "attribute")** approach might be a little quicker for setting individual attributes.

```
> myData = data.frame(cbind(            # Create data frame w/2 vectors
+   c(1, 0, 1, 1, 0),                   #   Vector 1
+   c(24, 38, 22, 51, 17)))             #   Vector 2

> # The attr approach
> attr(myData, "names") =               # Set col names to identify vars
+   c("sex", "age")
> attr(myData, "row.names") =           # Set row names to identify obs
+   c("Mary", "Mike", "Mia", "Mish", "Mark")
> myData                                # Display the data frame
     sex  age
Mary   1   24
Mike   0   38
Mia    1   22
Mish   1   51
Mark   0   17

> # The rownames/colnames approach
> rownames(myData) =                    # Add row names
+   c("Mary", "Mike", "Mia", "Mish", "Mark")
> colnames(myData) = c("sex", "age")    # Add column names
```

©SAGE Publications

This example shows two ways to set column and row names. Note the use of quotation marks for the attribute names in the **attr()** *approach. These approaches can also be used to set the row or column names to be equal to one of the existing rows or columns. For example,* **colnames(myData) = myData[1,]** *will set the column names to the values in row 1.*

You can erase an object's attributes individually by using **attr(object, "attribute") = NULL**. Alternatively, you can strip out all of the attributes with **attributes(object) = NULL**.

Another quick way to see the basic elements of an object is with the structure function: **str()**. It will give you a fast overview of what is in an object and the storage mode of the objects that make it up.

```
> str(myData)                        # Show data frame structure
'data.frame':                        5 obs. of  2 variables:
$ sex: num   1  0  1  1  0
$ age: num  24 38 22 51 17
```

OBJECTS AND ENVIRONMENTS

Before ending the discussion of R data object types, a word needs to be said about environments. An environment can be thought of as the space in which objects are defined. This usually won't be an issue for you, but it is important to be aware that R objects are stored within a specific environment. Most of your work will be in the "global environment," which is the overarching environment that is opened when you start R. Functions usually have their own environment, and thus, objects that are defined or manipulated within a function aren't necessarily available in the global environment.

This finishes our discussion of the basic R data object types: vectors, matrices, data frames, and lists. Our model, so far, is object types as

©SAGE Publications

containers that hold a group of objects that are characterized by their storage mode. The third element of this approach is object classes, which pass on more specific information about how an object should be treated.

R Object Classes

In addition to the storage mode and data type, most R objects have a class, which gives R more detailed information about how the object should be treated. Unlike our short lists of storage modes and basic object types, there are a very large number of possible classes. Indeed, you can even create your own classes. (But please don't!) A variety of R procedures use the class information to set up different behaviors. If you run a linear model with the **lm()**function, the output of that model will be a list. One of the elements of that list is the class, "lm," which tells R to deal with that object as the output of a linear model. The class of any object will be listed in the **attributes()**function and can also be accessed directly with the **class()**command.

```
> x = 5                             # Create an object
> class(x)                          # Show object class
[1] "numeric"
> class(mean)                       # Show class of existing R command
[1] "function"
> class(x) = "my made up class"     # Set new custom class for x
> attributes(x)                     # Show attributes of x
$class
[1] "my made up class"

> str(x)                            # Show structure of x
Class 'my made up class'  num 5
```

For the more generic commands like **summary()** or **print()**, which are almost always dependent on class information, you can use the **methods()** function to tell you which classes it will recognize. Here is a

©SAGE Publications

truncated list of the classes for which **summary()** has a distinct routine (there are 60 in all).

```
> methods(summary)
 [1] summary.aareg*          summary.agnes*
 [3] summary.aov             summary.aovlist
 [5] summary.areg.boot       summary.aspell*
 [7] summary.cch*            summary.clara*
 [9] summary.connection      summary.coxph*
[11] summary.coxph.penal*    summary.data.frame
```

You can also give methods the class information to see which generic functions are set up to work with them. For example, here is the list of functions that have a specialized approach to the results of a linear model (**class = lm**).

```
> methods(class = lm)
 [1] add1.lm*            alias.lm*          anova.lm
 [4] attrassign.lm*      case.names.lm*     confint.lm*
 [7] cooks.distance.lm*  deviance.lm*       dfbeta.lm*
[10] dfbetas.lm*         drop1.lm*          dummy.coef.lm*
[13] effects.lm*         extractAIC.lm*     family.lm*
[16] formula.lm*         hatvalues.lm       HTML.lm*
[19] influence.lm*       kappa.lm           labels.lm*
[22] logLik.lm*          model.frame.lm     model.matrix.lm
[25] nobs.lm*            plot.lm            predict.lm
[28] print.lm            proj.lm*           qr.lm*
[31] residuals.lm        rstandard.lm       rstudent.lm
[34] simulate.lm*        summary.lm         variable.names.lm*
[37] vcov.lm*

Nonvisible functions are indicated with an asterisk
```

We don't need to say much more about classes. They mostly work quietly in the background and don't cause too many problems. Most of what you have to know about objects and most of the more serious

©SAGE Publications

object problems will have to do with the basic object types and with confusion about the storage modes. This brings us back to our much delayed discussion of what I have called the two "pseudo storage modes."

The Pseudo Storage Modes

As indicated above, we need to deal with the complexities of fitting two other important kinds of objects into our schema: (1) date/time data and (2) factor data. Date and time values are obviously critical to many kinds of data projects. So too, "factor" is just the name R uses for categorical data, which is as common as it is critical.

The conceptually sensible home for both factors and date/time values is as storage modes. It makes sense to think of both as a characteristic of a data element, like the logical, numeric, and character storage modes. We would frequently expect to use dates or factors to fill one of our data containers. Indeed, we can use the **`c()`** function to create what look and behave like vectors of date/time or factor data, although R's **`is.vector()`** function won't officially recognize them as such. They can be packaged up and worked with as variables in a data frame. You can put them in a list.

The one container they don't fit into so well is matrices. If you put date/time or factor values into a matrix, they will be converted to a numeric storage mode, losing the critical information that makes date/time and categorical data distinctive. Mostly, we can work with date/time and factor values as if they were storage modes. But as we shall see, and as intimated by the way matrices refuse to acknowledge their character, dates and factors can cause significant problems if you aren't careful.

Date and Time as Storage Modes

I will spend just a little time on date/time data at this point. I've dedicated all of Chapter 9 to the issues you may confront when working with dates and times. Date and time values are obviously critically important for many data projects. Conceptually, the logical place for date and time values in our objects typology is as a storage mode. Date and time data can be thought of as individual elements that you would want to keep in a data container;

©SAGE Publications

some of your variables are numeric, some are character, and some are dates or times. It makes the most sense to think of date and time as a distinct storage mode, like logical, numeric, or character data.

Officially, R does not see dates or times this way. It stores dates as simple numeric values and then uses the class information to associate those values with instructions for how they should be interpreted to act and look like dates or times.

I'll go into much more detail on this process in Chapter 9. The one other thing to note now is that there are two kinds of date/time modes: (1) the Date mode and (2) the POSIX mode. The main difference is that POSIX values allow measurement down to even fractions of a second, while the Date mode only measures down to days. And POSIX values are more amenable to being broken up into component parts (e.g., days, minutes, or seconds).

Again, the important thing to keep in the back of your mind is that both POSIX and Date objects are technically stored as numeric data, with class and formatting information that translates them from raw numbers into recognizable date and time information.

Because date and time values are reasonably distinctive, these complexities are less likely to get you into trouble. You usually know when you are working with dates or times. Once you learn how R deals with them, as I'll cover in much more detail in Chapter 9, you should be able to manage them pretty well. Factors, I regret to say, can be a little trickier.

Factors

Factors are conceptually straightforward. They are just categorical variables. That means they are variables that can take on a discrete number of values or levels. They are very important in R because of their role in labeling, grouping, and sorting data and because of their ability to screw things up if you don't realize you are working with them.

As with dates, I think it makes the most conceptual sense to think of factors as a storage mode. A categorical variable is similar to the characterization of data as either numeric or character: A "factor" describes a unit of data that is stored as a categorical variable. This is not precisely how R sees it, which is why I have labeled this a pseudo storage mode.

Here is the critical thing: R really sees factors as two (or three) connected vectors (Figure 3.2). The first vector is a set of levels, which is all of

©SAGE Publications

the possible values the factor can take. The second is a set of pointers that tells R which of the levels to connect to each data point. Suppose we were categorizing things as made of metal, plastic, or wood. As a factor variable, this would have a vector of three levels ("metal", "plastic", and "wood") and a vector of pointers indicating which level to apply to each observation. The levels vector will be the length of the number of unique levels (three in this case). The pointer vector will be the length of the number of observations. Since in our current example there are three different levels, the pointer vector will have a 1, 2, or 3 for each observation. Problems arise when you get confused about when R is working with the levels and when it is working with the pointers. If you use **`typeof()`** on a factor, it will look at the vector of pointers and return "integer." **`attributes()`**, **`str()`**, and **`is.factor()`** are, therefore, more reliable tests for figuring out that you have a factor.

There is a third, optional, vector that is a set of labels to match up to the levels. The levels attribute is a vector of all the possible values that a factor variable can take. The labels attribute is a vector of labels corresponding to those levels. When you don't provide the labels vector, R just

Figure 3.2 The Structure of Factor Variables

The vector you see:	The vectors R sees: The Levels Vector	The Pointers Vector	The Labels Vector (Optional)
"Small"	L1: "L"	2	"Large"
"Small"	L2: "S"	2	"Small"
"Large"		1	
"Large"		1	
"Large"		1	
"Small"		2	
"Small"		2	

Note: This factor is created with the code: **`myFactor = factor(c("S", "S", "L", "L", "L", "S", "S"))`**. The labels are optional. Because the **`"ordered = TRUE"`** option isn't specified, the levels are simply sorted alphabetically.

©SAGE Publications

uses the levels as labels. If the names of your levels are self-explanatory, you don't need to add the labels vector. If, as in the following example, you would like to provide more explicit labels, you can include them as an option in the **factor()** command, and R will switch to using them instead of the original levels.

```
> c2ltr = c("FR", "UK", "SW",         # Create a variable with 2 letter
+   "NK", "SO")                       #   country abbreviations
> cname = c("France", "U.K.", "Sweden", # Create a variable with
+   "North Korea", "Somalia")         #   country names
> country = c("UK", "SW", "FR",       # Create a variable w/country codes
+   "SO", "NK")
> regime = c(rep("dem", 3),           #   and democratic status
+   rep("nondem", 2))
> nations.df =                        # Join into a data frame
+   data.frame(regime, country)
> nations.df                          # Print the new data frame
  regime country
1    dem      UK
2    dem      SW
3    dem      FR
4 nondem      SO
5 nondem      NK

> #now add labels
> nations.df$country =                # Create a country factor
+   factor(nations.df$country,
+     levels = c2ltr,                 # Connecting 2-letter codes
+     labels = cname)                 #   with country names

> nations.df                          # Print data frame
  regime     country
1    dem        U.K.
2    dem      Sweden
3    dem      France
4 nondem     Somalia
5 nondem North Korea
```

©SAGE Publications

This example shows the application of factor labels to a data frame.

As shown in Figure 3.2, R's default behavior is to sort factors alphabetically. In that case, we would probably rather list "Small" before "Large." We can exercise more control over how a factor gets set up by using the **levels=** and **labels=** options with the **factor()** command. The following example shows these procedures. Note in the second of these examples how the **labels=** option is dependent on the preexisting order of the levels and how it changes the levels themselves.

```
> # Set up the factor from figure 3.2
> myFactor = factor(c("S", "S", "L", "L", "L", "S", "S"))
> str(myFactor)                      # Show factor structure
Factor w/2 levels "L","S": 2 2 1 1 1 2 2

> # Change the ordering of the levels
> myFactor = factor(myFactor,          # Use values from current factor
+   levels = c("S", "L"))            # Specify the levels
> str(myFactor)                      # Show factor structure
Factor w/2 levels "S","L": 1 1 2 2 2 1 1

> myFactor = factor(myFactor,          # Use values from current factor
+   labels = c("Small", "Large"))    # Specify labels
> str(myFactor)                      # Show factor structure
Factor w/2 levels "Small","Large": 1 1 2 2 2 1 1
```

We can use the same **factor()** approach to add new levels to the factor. Or we can use the **levels()** function to do it more directly. Note in the second case how the **levels()** function can be used both to display and to modify the levels of the factor.

©SAGE Publications

```
> # Add additional level
> myFactor = factor(myFactor,         # Use values from current factor
+   levels =                          # Specify the levels w/addition
+   c("Small", "Medium", "Large"))
> str(myFactor)                       # Show factor structure
Factor w/3 levels "Small","Medium",..: 1 1 3 3 3 1 1

> levels(myFactor) =                  # Use levels() to add new level

+   c(levels(myFactor), "X-Large")    # Combine old levels with new

> summary(myFactor)                   # Summarize myFactor w/new levels

  Small  Medium   Large   X-Large

      4       0       3       0
```

Unfortunately, adding new observations to a factor is not as straightforward. With concatenation, R just tries to add to the pointers vector, and in so doing, it changes the storage mode from factor to the storage mode of whatever you are trying to add.

```
> myFactor2 = c(myFactor, "Medium")   # Can't concatenate w/new values
> myFactor2                           # Show result
[1] "1"  "1"  "3"  "3"  "3"  "1"  "1"  "Medium"

> myFactor2 = c(myFactor, 2)          # Can add to pointer vector
> myFactor2                           # But, dumps us out of factor mode
[1] 1 1 3 3 3 1 1 2
> is.factor(myFactor2)
[1] FALSE
```

©SAGE Publications

The odd trick here is that the data frame object type is smart enough to know how to add things to a factor when you use the row bind (**rbind()**) tool (we'll talk more about that in Chapters 4 and 10). This is an example of the class attribute working quietly and efficiently in the background. This data frame ability is a very good thing, since it means that factor data can be correctly handled when you are joining together more complex data sets. Here, it is in action.

```
> myDF = data.frame(myFactor)         # Put the factor into a data frame
> myDF = rbind(myDF, "Medium", "Small") # Add 2 new observations
> myDF                                # Show result
  myFactor
1    Small
2    Small
3    Large
4    Large
5    Large
6    Small
7    Small
8   Medium
9    Small

> myFactor2 = myDF$myFactor           # Return to vector status
> myFactor2                           # Confirm result
[1] Small  Small  Large  Large  Large  Small  Small  Medium Small
Levels: Small Medium Large
```

The **levels()** command can also be used to combine levels. Just duplicate the level names where you want to merge previously distinct levels, as in the following example:

```
> myFactor2                           # Display the factor
[1] Small  Small  Large  Large  Large  Small  Small  Medium Small
Levels: Small Medium Large
> levels(myFactor2)                   # Show the current levels
```

©SAGE Publications

```
[1] "Small"  "Medium" "Large"
> levels(myFactor2) =                # Modify the levels to combine
+   c("Small", "Large", "Large")     #   Medium with Large.
> myFactor2                          # Show the new version
[1] Small Small Large Large Large Small Small Large Small
Levels: Small Large
```

Finally, we should note that as intimated in Table 3.2, there are two kinds of factors: ordered and unordered. An ordered factor provides R with an explicit ordering for the different factor levels. To tell R to interpret your factor as ordered, you just include the **`ordered = TRUE`** option when setting up the factor, as shown in the following example:

```
> mySize = c("Small", "Medium",      # Create variable of all sizes
+   "Large", "X-Large")
> sort(mySize)                       # Sort (alphabetical default)
[1] "Large"   "Medium"  "Small"   "X-Large"
```

This regular sort is simply alphabetical.

```
> mySize = factor(mySize,            # Set as factor
+   levels = mySize,                 # Set factor levels from variable
+   ordered = T)                     # Make it an ordered factor

> sort(mySize)                       # Sort (now ordered)
[1] Small   Medium  Large   X-Large
Levels: Small < Medium < Large < X-Large
```

When we set it up as a factor and use the **`levels=`** *option, then R can sort in the desired order.*

```
> myData = c("Small", "Large",       # Here is some data w/sizes
+   "Small", "X-Large", "Medium")
```

©SAGE Publications

```
> sort(myData)                        # Sort (alphabetical default)
[1] "Large"   "Medium"  "Small"   "Small"   "X-Large"

> myData = factor(myData,             # Make it a factor
+   levels = mySize)                  # Use levels from mySize
> sort(myData)                        # Sort-now based on factor levels
[1] Small   Small   Medium  Large   X-Large
Levels: Small Medium Large X-Large
```

In this last example, we use our first factor as an ordered set of levels for sorting other variables.

You can already specify the order in which things are displayed by setting up the **levels=** option. There is a more important substantive distinction between ordered and unordered factors in that some basic statistical procedures, for example, **lm()** and **anova()**, make use of the ordering information and treat the factor quite differently (this gets into the statistics of contrasts).[6] We'll get into ordering and sorting of objects in much more detail in Chapter 6.

I'm sure, you are now thinking that these factors seem pretty useful and are wondering what the big deal is. And indeed, they are very useful. But here is why factors so often cause troubles: The *default* behavior in R is to turn character variables into factors whenever you import them into R or incorporate them into a data frame.[7] This makes it much more likely that you'll be caught unawares that what you think are characters R thinks are factors. There is a significant danger of getting confused between the factor, numeric, and character storage modes.

---

6. You may also encounter situations where you need to specify the first element in a factor to serve as the reference level. You can control this with the **relevel()** command setting the **ref=** option to the number of the level you want to serve as the reference level.

7. R does this because it is often much more efficient to store string variables as factors. In Appendix A, I show how to change this default behavior by using the **stringsAsFactors** option. This approach, however, comes at some cost both in efficiency and in reproducibility.

©SAGE Publications

Coercing Storage Modes

Before going further into the nature of the confusion that can arise between the factor, character, and numeric storage modes, it may be reassuring to say a little bit about our ability to coerce storage modes. Sometimes, you will want to force an object to retain its storage mode when incorporating it into a data frame. For example, you may want a set of character variables to retain their character mode rather than be forced into factors. To preserve a variable's mode, use the **`I()`** function. You can also use the **`as.`** modifier to force a type of object. In the following example, I create a vector that is turned into a vector of character elements because of the one clear bit of character data. You can see that prevents us from doing mathematical operations. I then force the vector to be numeric, which requires the character data to be left out as a missing value (**`NA`**).

```
> myVector = c(1, 15, 7, "Smith")       # Set up a vector
> typeof(myVector)                      # Show type for vector
[1] "character"

> myVector[2] + 1                       # Try math w/2nd element in vector
Error in myVector[2] + 1 : non-numeric argument to binary operator>

> typeof(myVector[2])                   # Type for 2nd element in vector
[1] "character"

> myVector = as.numeric(myVector)       # Vector forced to numeric

> myVector                              # Print vector
[1]  1 15  7 NA

> typeof(myVector)                      # Type for forced numeric vector
[1] "double"
```

In this example, we see that if any element in the vector is nonnumeric, R converts the whole vector to nonnumeric. If we force the vector to be numeric, the one nonnumeric element is turned to a missing value (NA). R will give us a warning that it was forced to create an NA value.

I should also mention here the **`asis()`** modifier. This modifies an object type so that it isn't transformed by certain operations. Most important,

©SAGE Publications

as we'll discuss in the next chapter, it can be used to prevent data frames from converting character or numeric data to factor data. This is part of a larger source of confusion that we need to address presently.

The Curse of Number/Character/Factor Confusion

One of the most confusing (and often frustrating) experiences is when you have character data that look like numbers. The character "7" is different from the number 7. If for some reason R has interpreted a variable as a set of characters, rather than as numbers, you need to be careful to transform it back to its numeric values before performing operations. Suppose you import the following data from a CSV (comma separated) file: 7, 8, missing, 8. R will put this into a vector. But recall that vectors can only hold things that are all of the same storage mode. Because of the word *missing*, R will force all of the data to be character data rather than letting the 7s and 8s be numbers.

```
> myData = c(7, 8, "missing", 8)        # Here we simulate the csv input
> sum(myData)                           # If we sum myData we get an error
Error in sum(myData) : invalid 'type' (character) of argument

> myData[1]                             # We see the value is a character
[1] "7"

> myData[1] + 2                         # Errors w/numeric operations
Error in myData[1] + 2 : non-numeric argument to binary operator

> as.numeric(myData[1])                 # Fix with transform to numeric
[1] 7

> as.numeric(myData[1]) + 2             # Now we can do numeric operation
[1] 9
```

Watch for the quotation marks, which are a dead giveaway that R thinks something is a character rather than a number.

©SAGE Publications

As shown in this example, you have to convert the character variable to numeric to use it as a number.

This problem becomes even more acute when working with factors. When R imports character data or incorporates them into a data frame, its default behavior is to convert it to a factor. R keeps track of factors as a set of values (the levels) and a set of pointers to those values. If you have a data set of kangaroos and koalas, the kangaroo/koala factor will have just two levels, “kangaroo” (1) and “koala” (2). The values in the factor itself will just be 1s or 2s to point to either “kangaroo” or “koala.” Here is an example:

```
> animal = (c(rep("kangas", 4),         # Create a character variable
+   rep("koalas", 5)))
> myData = data.frame(animal)           # Putting the data in a data frame
> levels(myData$animal)                 #  converts character to factor
[1] "kangas" "koalas"
> aninum = as.numeric(myData$animal)    # Create a numeric version
> myData = cbind(myData, aninum)        # Add that to the data frame
> myData                                # Show results
  animal  aninum
1 kangas      1
2 kangas      1
3 kangas      1
4 kangas      1
5 koalas      2
6 koalas      2
7 koalas      2
8 koalas      2
9 koalas      2
```

The **`levels()`** *function shows us that the animal variable has been converted to a factor with two levels.*

©SAGE Publications

This makes reasonable sense. The two levels of the factor are given the numbers 1 and 2. These are the pointers to the two levels of the factor. The problem comes if you have what you think are numbers but what R thinks are factor levels, as in the following example:

```
> myVar = (c(rep(7, 4),               # Create variable w/"missing"
+   "missing", rep(8, 5)))            #  value which forces to character
> myData = data.frame(myVar)          #  & becomes factor in data frame
> levels(myData$myVar)                # Show levels of unwanted factor
[1] "7"       "8"       "missing"

> myVar2 = as.numeric(myData$myVar)   # Convert to numeric
> myData = cbind(myData, myVar2)      # Add to data frame
> myData                              # Show data
     myVar  myVar2
1        7       1
2        7       1
3        7       1
4        7       1
5    missing     3
6        8       2
7        8       2
8        8       2
9        8       2
10       8       2
```

You might think these are numbers, but R treats them as factors because of the character value in observation 5.

As you can see, because of the one value with character data, R coerced all the data to character data. The data became a factor when placed into a data frame.[8] The factor has three levels: 7, 8, and "missing". When converted to numeric data, it is the pointers to those values (1, 2, and 3), rather than the interpreted values, that are used. If you try to do numeric operations on the **`as.numeric(myVar)`** values, you will be sorely disappointed.

---

8. If you are aware of the potential problem, you can prevent this transformation with the **`I()`** function for individual variables or the **`stringsAsFactors = F`** option for data files. See Appendix A for instructions on the use of this option.

©SAGE Publications

To convert factors to numeric, you cannot use **`as.numeric(myFactor)`**. That will just give you the values of the pointers. R Help (look up **`?as.factor`** to find this) recommends the following, somewhat cumbersome, approach:

```
as.numeric(levels(myFactor))[myFactor]
```

It is a little less efficient, but I think it a bit more intuitive to convert the factor to a character object and then to numeric. You can do this in two steps to keep everything very clear:

```
myTemp = as.character(myFactor)
myNum = as.numeric(myTemp)
```

Or you can put it all in one step:

```
myNum = as.numeric(as.character(myFactor))
```

Now that we have squared away the number/factor confusion, it's time to face up to the factor/character confusion. Here is where we really enter the twilight zone: You have to be careful about the distinction between factor and character variables.

Let's go back to our koalas and kangas data.

```
animal = (c(rep("kangas", 4), rep("koalas", 5)))
myData = data.frame(animal)
```

Let's find out what kinds of animals are in observation 4 and then change them to the other type.

```
> myData$animal[4]                    # Let's take a look at obs 4
[1] kangas
Levels: kangas koalas
```

©SAGE Publications

```
> myData$animal[4] = "koalas"          # Now we'll change it to "koalas"
> myData$animal[4]                     # Another look at num 4
[1] koalas
Levels: kangas koalas
```

Okay, no problem. Now let's try a different change.

```
> myData$animal[4] = "hippopotami"
Warning message:
In `[<-.factor`(`*tmp*`, 4, value = "hippopotami") :
  invalid factor level, NAs generated
```

Yikes! No hippopotami allowed! It isn't that R has some kind of marsupial filter. It's just that it kindly converted our character variable into a factor when we created the data frame. Once a factor is created, it can only deal with its existing levels. We can get around this in at least three ways (actually, since this is R, there are probably 47 ways around, but we'll stick to these 3).

First, we could have forced R to keep the animal variable as a character variable when setting up the data frame by using the **`I()`** function.

```
> myData = data.frame(I(animal))
> typeof(my.data$animal)               # confirm that animal is a character
variable
[1] "character"
> myData$animal[4]
[1] "kangas"
> myData$animal[4] = "hippopotami"
> myData$animal[4]
[1] "hippopotami"
```

Second, we could allow R to make the conversion to a factor, and then we could add another level to the factor.

©SAGE Publications

```
> animal = (c(rep("kangas", 4), rep("koalas", 5)))
> myData = data.frame(animal)           # Put the data in a data frame
> typeof(myData$animal)                 #  which converts it to a factor
[1] "integer"
```

Note: This is a bit misleading. This is a factor! So here R is just telling us about the vector of pointers.

```
> myData$animal = factor(myData$animal, # We'll add a new level to the mix
+   levels =                            #   with the levels option
+     c(levels(myData$animal),          # Combining old levels
+      "hippopotami"))                  #   with our new entry

> myData$animal[4]                      # Let's look at animal[4]
[1] kangas
Levels: kangas koalas hippopotami

> myData$animal[4] = "hippopotami"      # We can add "hippopotami" because
> myData$animal[4]                      #   that is included in the factor
[1] hippopotami
Levels: kangas koalas hippopotami
```

Finally, we could just make the conversions ourselves.

```
> animal = (c(rep("kangas", 4), rep("koalas", 5)))
> myData = data.frame(animal)           # Putting the data in a data frame
> typeof(myData$animal)                 # Check on the data type
[1] "integer"

> myData$animal =                       # Force variable back to character
+   as.character(myData$animal)
> typeof(myData$animal)                 # Recheck the type -- that works!
[1] "character"
```

©SAGE Publications

```
> myData$animal[4]                    # Show observation 4 value
[1] "kangas"

> myData$animal[4] = "hippopotami"    # Now make the change to obs 4
> myData$animal[4]
[1] "hippopotami"
```

Which of these approaches you choose will depend on your patience and on whether you want the variable to end up as a factor or a character vector.

Conclusion

I am sorry that we have had to spend so long disentangling R objects. Objects are at the core of how R works; building this foundation has been critical for everything that will follow. At the end of the day, the schema I have offered here, built on a mostly straightforward distinction between data object types, storage modes, and object classes, will carry you a very long way in the R world. This approach isn't quite regulation, but until you are ready to integrate your work in R with other high-level computer languages or need to worry about microsecond differences in efficiency, it will get you by.[8]

Every R data object, then, can be characterized by its type, its storage mode, and any class information that might be attached to it. You need only worry about the four data object types: vectors, matrices, data frames, and lists. The data stored within these containers will be characterized by one of five storage modes: logical, numeric, character, date/time, or factor. The last two of these are what I have called "pseudo storage modes." You have to be a little more careful when you encounter date/time or factor values. In Chapter 9, we'll get into the idiosyncrasies of date and time data. In the meantime, be particularly aware of where R has taken its own initiative to convert between characters, numbers, and factors without your explicit permission. Object classes shouldn't cause you much grief. When things aren't being displayed or processed as you would expect, you might use **`class()`** to check to be sure there isn't something amiss in the class information.

Data object types, storage modes, and classes. You've got this. And now we are ready to move on to the process of getting real data into R. Someday shortly beyond that, we'll get to the fun stuff.

8. It is interesting to note that official R ends up in this same place for reading in data. The read family of commands includes a **`colClasses=`** option, which lumps together date and factor variables with the more traditional logical, numeric, and character storage modes.

©SAGE Publications