# CHAPTER 5
# Data Structures

## Introduction

You can get pretty far in R just using vectors. That's what Chapter 2 is all about. This chapter moves beyond vectors to recipes for matrices, lists, factors, and data frames. If you have preconceptions about data structures, I suggest you put them aside. R does data structures differently.

If you want to study the technical aspects of R's data structures, I suggest reading *R in a Nutshell* (O'Reilly) and the *R Language Definition*. My notes here are more informal. These are things I wish I'd known when I started using R.

## Vectors

Here are some key properties of vectors:

*Vectors are homogeneous*
: All elements of a vector must have the same type or, in R terminology, the same mode.

*Vectors can be indexed by position*
: So `v[2]` refers to the second element of `v`.

*Vectors can be indexed by multiple positions, returning a subvector*
: So `v[c(2,3)]` is a subvector of `v` that consists of the second and third elements.

*Vector elements can have names*
: Vectors have a `names` property, the same length as the vector itself, that gives names to the elements:

```
> v <- c(10, 20, 30)
> names(v) <- c("Moe", "Larry", "Curly")
> print(v)
  Moe Larry Curly
   10    20    30
```

*If vector elements have names then you can select them by name*
Continuing the previous example:

```
> v["Larry"]
Larry
   20
```

## Lists

*Lists are heterogeneous*
Lists can contain elements of different types; in R terminology, list elements may have different modes. Lists can even contain other structured objects, such as lists and data frames; this allows you to create recursive data structures.

*Lists can be indexed by position*
So `lst[[2]]` refers to the second element of `lst`. Note the double square brackets.

*Lists let you extract sublists*
So `lst[c(2,3)]` is a sublist of `lst` that consists of the second and third elements. Note the single square brackets.

*List elements can have names*
Both `lst[["Moe"]]` and `lst$Moe` refer to the element named "Moe".

Since lists are heterogeneous and since their elements can be retrieved by name, a list is like a *dictionary* or *hash* or *lookup table* in other programming languages (Recipe 5.9). What's surprising (and cool) is that in R, unlike most of those other programming languages, lists can also be indexed by position.

## Mode: Physical Type

In R, every object has a *mode*, which indicates how it is stored in memory: as a number, as a character string, as a list of pointers to other objects, as a function, and so forth:

| Object | Example | Mode |
|---|---|---|
| Number | `3.1415` | numeric |
| Vector of numbers | `c(2.7.182, 3.1415)` | numeric |
| Character string | `"Moe"` | character |
| Vector of character strings | `c("Moe", "Larry", "Curly")` | character |
| Factor | `factor(c("NY", "CA", "IL"))` | numeric |
| List | `list("Moe", "Larry", "Curly")` | list |
| Data frame | `data.frame(x=1:3, y=c("NY", "CA", "IL"))` | list |
| Function | `print` | function |

The `mode` function gives us this information:

```
> mode(3.1415)                     # Mode of a number
[1] "numeric"
> mode(c(2.7182, 3.1415))          # Mode of a vector of numbers
[1] "numeric"
> mode("Moe")                      # Mode of a character string
[1] "character"
> mode(list("Moe","Larry","Curly")) # Mode of a list
[1] "list"
```

A critical difference between a vector and a list can be summed up this way:

- In a vector, all elements must have the same mode.
- In a list, the elements can have different modes.

## Class: Abstract Type

In R, every object also has a *class*, which defines its abstract type. The terminology is borrowed from object-oriented programming. A single number could represent many different things: a distance, a point in time, a weight. All those objects have a mode of "numeric" because they are stored as a number; but they could have different classes to indicate their interpretation.

For example, a `Date` object consists of a single number:

```
> d <- as.Date("2010-03-15")
> mode(d)
[1] "numeric"
> length(d)
[1] 1
```

But it has a class of `Date`, telling us how to interpret that number; namely, as the number of days since January 1, 1970:

```
> class(d)
[1] "Date"
```

R uses an object's class to decide how to process the object. For example, the generic function `print` has specialized versions (called *methods*) for printing objects according to their class: `data.frame`, `Date`, `lm`, and so forth. When you print an object, R calls the appropriate `print` function according to the object's class.

## Scalars

The quirky thing about scalars is their relationship to vectors. In some software, scalars and vectors are two different things. In R, they are the same thing: a scalar is simply a vector that contains exactly one element. In this book I often use the term "scalar", but that's just shorthand for "vector with one element."

Consider the built-in constant `pi`. It is a scalar:

```
> pi
[1] 3.141593
```

Since a scalar is a one-element vector, you can use vector functions on `pi`:

```
> length(pi)
[1] 1
```

You can index it. The first (and only) element is π, of course:

```
> pi[1]
[1] 3.141593
```

If you ask for the second element, there is none:

```
> pi[2]
[1] NA
```

## Matrices

In R, a matrix is just a vector that has dimensions. It may seem strange at first, but you can transform a vector into a matrix simply by giving it dimensions.

A vector has an attribute called `dim`, which is initially `NULL`, as shown here:

```
> A <- 1:6
> dim(A)
NULL
> print(A)
[1] 1 2 3 4 5 6
```

We give dimensions to the vector when we set its `dim` attribute. Watch what happens when we set our vector dimensions to 2 × 3 and print it:

```
> dim(A) <- c(2,3)
> print(A)
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
```

Voilà! The vector was reshaped into a 2 × 3 matrix.

A matrix can be created from a list, too. Like a vector, a list has a `dim` attribute, which is initially `NULL`:

```
> B <- list(1,2,3,4,5,6)
> dim(B)
NULL
```

If we set the `dim` attribute, it gives the list a shape:

```
> dim(B) <- c(2,3)
> print(B)
     [,1] [,2] [,3]
[1,] 1    3    5
[2,] 2    4    6
```

Voilà! We have turned this list into a 2 × 3 matrix.

## Arrays

The discussion of matrices can be generalized to 3-dimensional or even *n*-dimensional structures: just assign more dimensions to the underlying vector (or list). The following example creates a 3-dimensional array with dimensions 2 × 3 × 2:

```
> D <- 1:12
> dim(D) <- c(2,3,2)
> print(D)
, , 1

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

, , 2

     [,1] [,2] [,3]
[1,]    7    9   11
[2,]    8   10   12
```

Note that R prints one "slice" of the structure at a time, since it's not possible to print a 3-dimensional structure on a 2-dimensional medium.

### Matrices Made from Lists

It strikes me as very odd that we can turn a list into a matrix just by giving the list a `dim` attribute. But wait; it gets stranger.

Recall that a list can be heterogeneous (mixed modes). We can start with a heterogeneous list, give it dimensions, and thus create a heterogeneous matrix. This code snippet creates a matrix that is a mix of numeric and character data:

```
> C <- list(1, 2, 3, "X", "Y", "Z")
> dim(C) <- c(2,3)
> print(C)
     [,1] [,2] [,3]
[1,] 1    3    "Y"
[2,] 2    "X"  "Z"
```

To me this is strange because I ordinarily assume a matrix is purely numeric, not mixed. R is not that restrictive.

The possibility of a heterogeneous matrix may seem powerful and strangely fascinating. However, it creates problems when you are doing normal, day-to-day stuff with matrices. For example, what happens when the matrix C (above) is used in matrix multiplication? What happens if it is converted to a data frame? The answer is that odd things happen.

In this book, I generally ignore the pathological case of a heterogeneous matrix. I assume you've got simple, vanilla matrices. Some recipes involving matrices may work oddly (or not at all) if your matrix contains mixed data. Converting such a matrix to a vector or data frame, for instance, can be problematic (Recipe 5.33).

## Factors

A factor looks like a vector, but it has special properties. R keeps track of the unique values in a vector, and each unique value is called a *level* of the associated factor. R uses a compact representation for factors, which makes them efficient for storage in data frames. In other programming languages, a factor would be represented by a *vector of enumerated values*.

There are two key uses for factors:

*Categorical variables*
: A factor can represent a categorical variable. Categorical variables are used in contingency tables, linear regression, analysis of variance (ANOVA), logistic regression, and many other areas.

*Grouping*
: This is a technique for labeling or tagging your data items according to their group. See the "Introduction" to Chapter 6.

## Data Frames

A data frame is powerful and flexible structure. Most serious R applications involve data frames. A data frame is intended to mimic a dataset, such as one you might encounter in SAS or SPSS.

A data frame is a tabular (rectangular) data structure, which means that it has rows and columns. It is not implemented by a matrix, however. Rather, a data frame is a list:

- The elements of the list are vectors and/or factors.*
- Those vectors and factors are the columns of the data frame.
- The vectors and factors must all have the same length; in other words, all columns must have the same height.
- The equal-height columns give a rectangular shape to the data frame.
- The columns must have names.

Because a data frame is both a list and a rectangular structure, R provides two different paradigms for accessing its contents:

- You can use list operators to extract columns from a data frame, such as `dfrm[i]`, `dfrm[[i]]`, or `dfrm$`*`name`*.
- You can use matrix-like notation, such as `dfrm[i,j]`, `dfrm[i,]`, or `dfrm[,j]`.

Your perception of a data frame likely depends on your background:

* A data frame can be built from a mixture of vectors, factors, and matrices. The columns of the matrices become columns in the data frame. The number of *rows* in each matrix must match the *length* of the vectors and factors. In other words, all elements of a data frame must have the same *height*.

*To a statistician*
: A data frame is a table of observations. Each row contains one observation. Each observation must contain the same variables. These variables are called columns, and you can refer to them by name. You can also refer to the contents by row number and column number, just as with a matrix.

*To a SQL programmer*
: A data frame is a table. The table resides entirely in memory, but you can save it to a flat file and restore it later. You needn't declare the column types because R figures that out for you.

*To an Excel user*
: A data frame is like a worksheet, or perhaps a range within a worksheet. It is more restrictive, however, in that each column has a type.

*To an SAS user*
: A data frame is like a SAS dataset for which all the data resides in memory. R can read and write the data frame to disk, but the data frame must be in memory while R is processing it.

*To an R programmer*
: A data frame is a hybrid data structure, part matrix and part list. A column can contain numbers, character strings, or factors but not a mix of them. You can index the data frame just like you index a matrix. The data frame is also a list, where the list elements are the columns, so you can access columns by using list operators.

*To a computer scientist*
: A data frame is a rectangular data structure. The columns are strongly typed, and each column must be numeric values, character strings, or a factor. Columns must have labels; rows may have labels. The table can be indexed by position, column name, and/or row name. It can also be accessed by list operators, in which case R treats the data frame as a list whose elements are the columns of the data frame.

*To an executive*
: `You can put names and numbers into a data frame. It's easy! A data frame is like a little database. Your staff will enjoy using data frames.`

## 5.1 Appending Data to a Vector

### Problem

You want to append additional data items to a vector.

### Solution

Use the vector constructor (`c`) to construct a vector with the additional data items:

```
> v <- c(v,newItems)
```

For a single item, you can also assign the new item to the next vector element. R will automatically extend the vector:

```
> v[length(v)+1] <- newItem
```

## Discussion

If you ask me about appending a data item to a vector, I will suggest that maybe you shouldn't.

R works best when you think about entire vectors, not single data items. Are you repeatedly appending items to a vector? If so, then you are probably working inside a loop. That's OK for small vectors, but for large vectors your program will run slowly. The memory management in R works poorly when you repeatedly extend a vector by one element. Try to replace that loop with vector-level operations. You'll write less code, and R will run much faster.

Nonetheless, one does occasionally need to append data to vectors. My experiments show that the most efficient way is to create a new vector using the vector constructor (`c`) to join the old and new data. This works for appending single elements or multiple elements:

```
> v <- c(1,2,3)
> v <- c(v,4)          # Append a single value to v
> v
[1] 1 2 3 4
> w <- c(5,6,7,8)
> v <- c(v,w)          # Append an entire vector to v
> v
[1] 1 2 3 4 5 6 7 8
```

You can also append an item by assigning it to the position past the end of the vector, as shown in the Solution. In fact, R is very liberal about extending vectors. You can assign to any element and R will expand the vector to accommodate your request:

```
> v <- c(1,2,3)        # Create a vector of three elements
> v[10] <- 10          # Assign to the 10th element
> v                    # R extends the vector automatically
 [1]  1  2  3 NA NA NA NA NA NA 10
```

Note that R did not complain about the out-of-bounds subscript. It just extended the vector to the needed length, filling with NA.

R includes an `append` function that creates a new vector by appending items to an existing vector. However, my experiments show that this function runs more slowly than both the vector constructor and the element assignment.

## 5.2 Inserting Data into a Vector

### Problem

You want to insert one or more data items into a vector.

### Solution

Despite its name, the `append` function inserts data into a vector by using the `after` parameter, which gives the insertion point for the new item or items:

```
> append(vec, newvalues, after=n)
```

### Discussion

The new items will be inserted at the position given by `after`. This example inserts 99 into the middle of a sequence:

```
> append(1:10, 99, after=5)
 [1]  1  2  3  4  5 99  6  7  8  9 10
```

The special value of `after=0` means insert the new items at the head of the vector:

```
> append(1:10, 99, after=0)
 [1] 99  1  2  3  4  5  6  7  8  9 10
```

The comments in Recipe 5.1 apply here, too. If you are inserting single items into a vector, you might be working at the element level when working at the vector level would be easier to code and faster to run.

## 5.3 Understanding the Recycling Rule

### Problem

You want to understand the mysterious Recycling Rule that governs how R handles vectors of unequal length.

### Discussion

When you do vector arithmetic, R performs element-by-element operations. That works well when both vectors have the same length: R pairs the elements of the vectors and applies the operation to those pairs.

But what happens when the vectors have unequal lengths?

In that case, R invokes the Recycling Rule. It processes the vector element in pairs, starting at the first elements of both vectors. At a certain point, the shorter vector is exhausted while the longer vector still has unprocessed elements. R returns to the beginning of the shorter vector, “recycling” its elements; continues taking elements from the longer vector; and completes the operation. It will recycle the shorter-vector elements as often as necessary until the operation is complete.

It's useful to visualize the Recycling Rule. Here is a diagram of two vectors, 1:6 and 1:3:

| 1:6 | 1:3 |
|---|---|
| 1 | 1 |
| 2 | 2 |
| 3 | 3 |
| 4 | |
| 5 | |
| 6 | |

Obviously, the 1:6 vector is longer than the 1:3 vector. If we try to add the vectors using (1:6) + (1:3), it appears that 1:3 has too few elements. However, R recycles the elements of 1:3, pairing the two vectors like this and producing a six-element vector:

| 1:6 | 1:3 | (1:6) + (1:3) |
|---|---|---|
| 1 | 1 | 2 |
| 2 | 2 | 4 |
| 3 | 3 | 6 |
| 4 | *1* | 5 |
| 5 | *2* | 7 |
| 6 | *3* | 9 |

Here is what you see at the command line:

```
> (1:6) + (1:3)
[1] 2 4 6 5 7 9
```

It's not only vector operations that invoke the Recycling Rule; functions can, too. The `cbind` function can create column vectors, such as the following column vectors of 1:6 and 1:3. The two column have different heights, of course:

```
> cbind(1:6)
     [,1]
[1,]    1
[2,]    2
[3,]    3
[4,]    4
[5,]    5
[6,]    6
```

```
> cbind(1:3)
     [,1]
[1,]    1
[2,]    2
[3,]    3
```

If we try binding these column vectors together into a two-column matrix, the lengths are mismatched. The 1:3 vector is too short, so `cbind` invokes the Recycling Rule and recycles the elements of 1:3:

```
> cbind(1:6, 1:3)
     [,1] [,2]
[1,]    1    1
[2,]    2    2
[3,]    3    3
[4,]    4    1
[5,]    5    2
[6,]    6    3
```

If the longer vector's length is not a multiple of the shorter vector's length, R gives a warning. That's good, since the operation is highly suspect and there is likely a bug in your logic:

```
> (1:6) + (1:5)          # Oops! 1:5 is one element too short
[1]  2  4  6  8 10  7
Warning message:
In (1:6) + (1:5) :
  longer object length is not a multiple of shorter object length
```

Once you understand the Recycling Rule, you will realize that operations between a vector and a scalar are simply applications of that rule. In this example, the 10 is recycled repeatedly until the vector addition is complete:

```
> (1:6) + 10
[1] 11 12 13 14 15 16
```

## 5.4 Creating a Factor (Categorical Variable)

### Problem

You have a vector of character strings or integers. You want R to treat them as a factor, which is R's term for a categorical variable.

### Solution

The `factor` function encodes your vector of discrete values into a factor:

```
> f <- factor(v)          # v is a vector of strings or integers
```

If your vector contains only a subset of possible values and not the entire universe, then include a second argument that gives the possible levels of the factor:

```
> f <- factor(v, levels)
```

## Discussion

In R, each possible value of a categorical variable is called a *level*. A vector of levels is called a *factor*. Factors fit very cleanly into the vector orientation of R, and they are used in powerful ways for processing data and building statistical models.

Most of the time, converting your categorical data into a factor is a simple matter of calling the `factor` function, which identifies the distinct levels of the categorical data and packs them into a factor:

```
> f <- factor(c("Win","Win","Lose","Tie","Win","Lose"))
> f
[1] Win  Win  Lose Tie  Win  Lose
Levels: Lose Tie Win
```

Notice that when we printed the factor, `f`, R did not put quotes around the values. They are levels, not strings. Also notice that when we printed the factor, R also displayed the distinct levels below the factor.

If your vector contains only a subset of all the possible levels, then R will have an incomplete picture of the possible levels. Suppose you have a string-valued variable `wday` that gives the day of the week on which your data was observed:

```
> f <- factor(wday)
> f
 [1] Wed Thu Mon Wed Thu Thu Thu Tue Thu Tue
Levels: Mon Thu Tue Wed
```

R thinks that Monday, Thursday, Tuesday, and Wednesday are the only possible levels. Friday is not listed. Apparently, the lab staff never made observations on Friday, so R does not know that Friday is a possible value. Hence you need to list the possible levels of `wday` explicitly:

```
> f <- factor(wday, c("Mon","Tue","Wed","Thu","Fri"))
> f
 [1] Wed Thu Mon Wed Thu Thu Thu Tue Thu Tue
Levels: Mon Tue Wed Thu Fri
```

Now R understands that `f` is a factor with five possible levels. It knows their correct order, too. It originally put Thursday before Tuesday because it assumes alphabetical order by default.† The explicit second argument defines the correct order.

In many situations it is not necessary to call `factor` explicitly. When an R function requires a factor, it usually converts your data to a factor automatically. The `table` function, for instance, works only on factors, so it routinely converts its inputs to factors without asking. You must explicitly create a factor variable when you want to specify the full set of levels or when you want to control the ordering of levels.

† More precisely, it orders the names according to your Locale.

### See Also

See Recipe 12.6 to create a factor from continuous data.

## 5.5 Combining Multiple Vectors into One Vector and a Factor

### Problem

You have several groups of data, with one vector for each group. You want to combine the vectors into one large vector and simultaneously create a parallel factor that identifies each value's original group.

### Solution

Create a list that contains the vectors. Use the `stack` function to combine the list into a two-column data frame:

```
> comb <- stack(list(v1=v1, v2=v2, v3=v3))     # Combine 3 vectors
```

The data frame's columns are called `values` and `ind`. The first column contains the data, and the second column contains the parallel factor.

### Discussion

Why in the world would you want to mash all your data into one big vector and a parallel factor? The reason is that many important statistical functions require the data in that format.

Suppose you survey freshmen, sophomores, and juniors regarding their confidence level ("What percentage of the time do you feel confident in school?"). Now you have three vectors, called `freshmen`, `sophomores`, and `juniors`. You want to perform an ANOVA analysis of the differences between the groups. The ANOVA function, `aov`, requires one vector with the survey results as well as a parallel factor that identifies the group. You can combine the groups using the `stack` function:

```
> comb <- stack(list(fresh=freshmen, soph=sophomores, jrs=juniors))
> print(comb)
   values   ind
1    0.60 fresh
2    0.35 fresh
3    0.44 fresh
4    0.62 fresh
5    0.60 fresh
6    0.70  soph
7    0.61  soph
8    0.63  soph
9    0.87  soph
10   0.85  soph
11   0.70  soph
12   0.64  soph
```

```
13   0.76   jrs
14   0.71   jrs
15   0.92   jrs
16   0.87   jrs
```

Now you can perform the ANOVA analysis on the two columns:

```
> aov(values ~ ind, data=comb)
```

When building the list we must provide tags for the list elements (the tags are `fresh`, `soph`, and `jrs` in this example). Those tags are required because `stack` uses them as the levels of the parallel factor.

## 5.6 Creating a List

### Problem

You want to create and populate a list.

### Solution

To create a list from individual data items, use the `list` function:

```
> lst <- list(x, y, z)
```

### Discussion

Lists can be quite simple, such as this list of three numbers:

```
> lst <- list(0.5, 0.841, 0.977)
> lst
[[1]]
[1] 0.5

[[2]]
[1] 0.841

[[3]]
[1] 0.977
```

When R prints the list, it identifies each list element by its position (`[[1]]`, `[[2]]`, `[[3]]`) and prints the element's value (e.g., `[1] 0.5`) under its position.

More usefully, lists can—unlike vectors—contain elements of different modes (types). Here is an extreme example of a mongrel created from a scalar, a character string, a vector, and a function:

```
> lst <- list(3.14, "Moe", c(1,1,2,3), mean)
> lst
[[1]]
[1] 3.14

[[2]]
```

```
[1] "Moe"

[[3]]
[1] 1 1 2 3

[[4]]
function (x, ...)
UseMethod("mean")
<environment: namespace:base>
```

You can also build a list by creating an empty list and populating it. Here is our "mongrel" example built in that way:

```
> lst <- list()
> lst[[1]] <- 3.14
> lst[[2]] <- "Moe"
> lst[[3]] <- c(1,1,2,3)
> lst[[4]] <- mean
```

List elements can be named. The `list` function lets you supply a name for every element:

```
> lst <- list(mid=0.5, right=0.841, far.right=0.977)
> lst
$mid
[1] 0.5

$right
[1] 0.841

$far.right
[1] 0.977
```

### See Also

See the "Introduction" to this chapter for more about lists; see Recipe 5.9 for more about building and using lists with named elements.

## 5.7 Selecting List Elements by Position

### Problem

You want to access list elements by position.

### Solution

Use one of these ways. Here, `lst` is a list variable:

`lst[[`*n*`]]`
: Select the *n*th element from the list.

`lst[c(`$n_1$`, `$n_2$`, ..., `$n_k$`)]`
: Returns a list of elements, selected by their positions.

Note that the first form returns a single element and the second returns a list.

## Discussion

Suppose we have a list of four integers, called `years`:

```
> years <- list(1960, 1964, 1976, 1994)
> years
[[1]]
[1] 1960

[[2]]
[1] 1964

[[3]]
[1] 1976

[[4]]
[1] 1994
```

We can access single elements using the double-square-bracket syntax:

```
> years[[1]]
[1] 1960
```

We can extract sublists using the single-square-bracket syntax:

```
> years[c(1,2)]
[[1]]
[1] 1960

[[2]]
[1] 1964
```

This syntax can be confusing because of a subtlety: there is an important difference between `lst[[`*n*`]]` and `lst[`*n*`]`. They are not the same thing:

`lst[[`*n*`]]`
: This is an element, not a list. It is the *n*th element of `lst`.

`lst[`*n*`]`
: This is a list, not an element. The list contains one element, taken from the *n*th element of `lst`. This is a special case of `lst[c(`$n_1$`, `$n_2$`, ..., `$n_k$`)]` in which we eliminated the `c(...)` construct because there is only one `n`.

The difference becomes apparent when we inspect the structure of the result—one is a number; the other is a list:

```
> class(years[[1]])
[1] "numeric"

> class(years[1])
[1] "list"
```

The difference becomes annoyingly apparent when we `cat` the value. Recall that `cat` can print atomic values or vectors but complains about printing structured objects:

```
> cat(years[[1]], "\n")
1960
> cat(years[1], "\n")
Error in cat(list(...), file, sep, fill, labels, append) :
  argument 1 (type 'list') cannot be handled by 'cat'
```

We got lucky here because R alerted us to the problem. In other contexts, you might work long and hard to figure out that you accessed a sublist when you wanted an element, or vice versa.

## 5.8 Selecting List Elements by Name

### Problem

You want to access list elements by their names.

### Solution

Use one of these forms. Here, `lst` is a list variable:

`lst[["`*`name`*`"]]`
: Selects the element called *name*. Returns `NULL` if no element has that name.

`lst$`*`name`*
: Same as previous, just different syntax.

`lst[c(`$name_1$`, `$name_2$`, ..., `$name_k$`)]`
: Returns a list built from the indicated elements of `lst`.

Note that the first two forms return an element whereas the third form returns a list.

### Discussion

Each element of a list can have a name. If named, the element can be selected by its name. This assignment creates a list of four named integers:

```
> years <- list(Kennedy=1960, Johnson=1964, Carter=1976, Clinton=1994)
```

These next two expressions return the same value—namely, the element that is named "Kennedy":

```
> years[["Kennedy"]]
[1] 1960
> years$Kennedy
[1] 1960
```

The following two expressions return sublists extracted from `years`:

```
> years[c("Kennedy","Johnson")]
$Kennedy
```

```
[1] 1960

$Johnson
[1] 1964

> years["Carter"]
$Carter
[1] 1976
```

Just as with selecting list elements by position (Recipe 5.7), there is an important difference between `lst[["name"]]` and `lst["name"]`. They are not the same:

`lst[["name"]]`
: This is an element, not a list.

`lst["name"]`
: This is a list, not an element. This is a special case of `lst[c(`*name*$_1$`,` *name*$_2$`, ...,` *name*$_k$`)]` in which we don't need the `c(...)` construct because there is only one name.

### See Also

See Recipe 5.7 to access elements by position rather than by name.

## 5.9 Building a Name/Value Association List

### Problem

You want to create a list that associates names and values—as would a dictionary, hash, or lookup table in another programming language.

### Solution

The `list` function lets you give names to elements, creating an association between each name and its value:

```
> lst <- list(mid=0.5, right=0.841, far.right=0.977)
```

If you have parallel vectors of names and values, you can create an empty list and then populate the list by using a vectorized assignment statement:

```
> lst <- list()
> lst[names] <- values
```

### Discussion

Each element of a list can be named, and you can retrieve list elements by name. This gives you a basic programming tool: the ability to associate names with values.

You can assign element names when you build the list. The `list` function allows arguments of the form *`name=value`*:

```
> lst <- list(
+           far.left=0.023,
+           left=0.159,
+           mid=0.500,
+           right=0.841,
+           far.right=0.977)
> lst
$far.left
[1] 0.023

$left
[1] 0.159

$mid
[1] 0.5

$right
[1] 0.841

$far.right
[1] 0.977
```

One way to name the elements is to create an empty list and then populate it via assignment statements:

```
> lst <- list()
> lst$far.left <- 0.023
> lst$left <- 0.159
> lst$mid <- 0.500
> lst$right <- 0.841
> lst$far.right <- 0.977
```

Sometimes you have a vector of names and a vector of corresponding values:

```
> values <- pnorm(-2:2)
> names <- c("far.left", "left", "mid", "right", "far.right")
```

You can associate the names and the values by creating an empty list and then populating it with a vectorized assignment statement:

```
> lst <- list()
> lst[names] <- values
> lst
$far.left
[1] 0.02275013

$left
[1] 0.1586553

$mid
[1] 0.5

$right
[1] 0.8413447

$far.right
[1] 0.9772499
```

Once the association is made, the list can "translate" names into values through a simple list lookup:

```
> cat("The left limit is", lst[["left"]], "\n")
The left limit is 0.1586553
> cat("The right limit is", lst[["right"]], "\n")
The right limit is 0.8413447

> for (nm in names(lst)) cat("The", nm, "limit is", lst[[nm]], "\n")
The far.left limit is 0.02275013
The left limit is 0.1586553
The mid limit is 0.5
The right limit is 0.8413447
The far.right limit is 0.9772499
```

## 5.10 Removing an Element from a List

### Problem

You want to remove an element from a list.

### Solution

Assign `NULL` to the element. R will remove it from the list.

### Discussion

To remove a list element, select it by position or by name, and then assign `NULL` to the selected element:

```
> years
$Kennedy
[1] 1960

$Johnson
[1] 1964

$Carter
[1] 1976

$Clinton
[1] 1994

> years[["Johnson"]] <- NULL              # Remove the element labeled "Johnson"
> years
$Kennedy
[1] 1960

$Carter
[1] 1976

$Clinton
[1] 1994
```

You can remove multiple elements this way, too:

```
> years[c("Carter","Clinton")] <- NULL     # Remove two elements
> years
$Kennedy
[1] 1960
```

## 5.11 Flatten a List into a Vector

### Problem

You want to flatten all the elements of a list into a vector.

### Solution

Use the `unlist` function.

### Discussion

There are many contexts that require a vector. Basic statistical functions work on vectors but not on lists, for example. If `iq.scores` is a list of numbers, then we cannot directly compute their mean:

```
> mean(iq.scores)
[1] NA
Warning message:
In mean.default(iq.scores) :
  argument is not numeric or logical: returning NA
```

Instead, we must flatten the list into a vector using `unlist` and then compute the mean of the result:

```
> mean(unlist(iq.scores))
[1] 106.4452
```

Here is another example. We can `cat` scalars and vectors, but we cannot `cat` a list:

```
> cat(iq.scores, "\n")
Error in cat(list(...), file, sep, fill, labels, append) :
  argument 1 (type 'list') cannot be handled by 'cat'
```

One solution is to flatten the list into a vector before printing:

```
> cat("IQ Scores:", unlist(iq.scores), "\n")
IQ Scores: 89.73383 116.5565 113.0454
```

### See Also

Conversions such as this are discussed more fully in Recipe 5.33.