VECTORS

The fundamental data type in R is the vector. You saw a few examples in Chapter 1, and now you’ll learn the details. We’ll start by examining how vectors relate to some other data types in R. You’ll see that unlike in languages in the C family, individual numbers (scalars) do not have separate data types but instead are special cases of vectors. On the other hand, as in C family languages, matrices are special cases of vectors.

We’ll spend a considerable amount of time on the following topics:

- **Recycling**  The automatic lengthening of vectors in certain settings
- **Filtering**   The extraction of subsets of vectors
- **Vectorization**  Where functions are applied element-wise to vectors

All of these operations are central to R programming, and you will see them referred to often in the remainder of the book.
2.1 Scalars, Vectors, Arrays, and Matrices

In many programming languages, vector variables are considered different from scalars, which are single-number variables. Consider the following C code, for example:

```c
int x;
int y[3];
```

This requests the compiler to allocate space for a single integer named x and a three-element integer array (C terminology analogous to R’s vector type) named y. But in R, numbers are actually considered one-element vectors, and there is really no such thing as a scalar.

R variable types are called modes. Recall from Chapter 1 that all elements in a vector must have the same mode, which can be integer, numeric (floating-point number), character (string), logical (Boolean), complex, and so on. If you need your program code to check the mode of a variable x, you can query it by the call `typeof(x)`.

Unlike vector indices in ALGOL-family languages, such as C and Python, vector indices in R begin at 1.

### 2.1.1 Adding and Deleting Vector Elements

Vectors are stored like arrays in C, contiguously, and thus you cannot insert or delete elements—something you may be used to if you are a Python programmer. The size of a vector is determined at its creation, so if you wish to add or delete elements, you’ll need to reassign the vector.

For example, let’s add an element to the middle of a four-element vector:

```r
> x <- c(88, 5, 12, 13)
> x <- c(x[1:3], 168, x[4]) # insert 168 before the 13
> x
[1] 88 5 12 168 13
```

Here, we created a four-element vector and assigned it to `x`. To insert a new number 168 between the third and fourth elements, we strung together the first three elements of `x`, then the 168, then the fourth element of `x`. This creates a new five-element vector, leaving `x` intact for the time being. We then assigned that new vector to `x`.

In the result, it appears as if we had actually changed the vector stored in `x`, but really we created a new vector and stored that vector in `x`. This difference may seem subtle, but it has implications. For instance, in some cases, it may restrict the potential for fast performance in R, as discussed in Chapter 14.

**NOTE**  
For readers with a background in C, internally, `x` is really a pointer, and the reassignment is implemented by pointing `x` to the newly created vector.
2.1.2 Obtaining the Length of a Vector

You can obtain the length of a vector by using the `length()` function:

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

In this example, we already know the length of `x`, so there really is no need to query it. But in writing general function code, you'll often need to know the lengths of vector arguments.

For instance, suppose that we wish to have a function that determines the index of the first 1 value in the function's vector argument (assuming we are sure there is such a value). Here is one (not necessarily efficient) way we could write the code:

```r
first1 <- function(x) {
  for (i in 1:length(x)) {
    if (x[i] == 1) break  # break out of loop
  }
  return(i)
}
```

Without the `length()` function, we would have needed to add a second argument to `first1()`, say naming it `n`, to specify the length of `x`.

Note that in this case, writing the loop as follows won't work:

```r
for (n in x)
```

The problem with this approach is that it doesn't allow us to retrieve the index of the desired element. Thus, we need an explicit loop, which in turn requires calculating the length of `x`.

One more point about that loop: For careful coding, you should worry that `length(x)` might be 0. In such a case, look what happens to the expression `1:length(x)` in our `for` loop:

```r
> x <- c()
> x
NULL
> length(x)
[1] 0
> 1:length(x)
[1] 1 0
```

Our variable `i` in this loop takes on the value 1, then 0, which is certainly not what we want if the vector `x` is empty.

A safe alternative is to use the more advanced R function `seq()`, as we'll discuss in Section 2.4.4.
2.1.3 Matrices and Arrays as Vectors

Arrays and matrices (and even lists, in a sense) are actually vectors too, as you’ll see. They merely have extra class attributes. For example, matrices have the number of rows and columns. We’ll discuss them in detail in the next chapter, but it’s worth noting now that arrays and matrices are vectors, and that means that everything we say about vectors applies to them, too.

Consider the following example:

```r
> m
[,1] [,2]
[1,] 1 2
[2,] 3 4
> m + 10:13
[,1] [,2]
[1,] 11 14
[2,] 14 17
```

The 2-by-2 matrix `m` is stored as a four-element vector, column-wise, as (1,3,2,4). We then added (10,11,12,13) to it, yielding (11,14,14,17), but R remembered that we were working with matrices and thus gave the 2-by-2 result you see in the example.

2.2 Declarations

Typically, compiled languages require that you declare variables; that is, warn the interpreter/compiler of the variables’ existence before using them. This is the case in our earlier C example:

```c
int x;
int y[3];
```

As with most scripting languages (such as Python and Perl), you do not declare variables in R. For instance, consider this code:

```r
z <- 3
```

This code, with no previous reference to `z`, is perfectly legal (and commonplace).

However, if you reference specific elements of a vector, you must warn R. For instance, say we wish `y` to be a two-component vector with values 5 and 12. The following will not work:

```r
> y[1] <- 5
> y[2] <- 12
```
Instead, you must create `y` first, for instance this way:

```r
> y <- vector(length=2)
> y[1] <- 5
> y[2] <- 12
```

The following will also work:

```r
> y <- c(5,12)
```

This approach is all right because on the right-hand side we are creating a new vector, to which we then bind `y`.

The reason we cannot suddenly spring an expression like `y[2]` on `R` stems from `R`'s functional language nature. The reading and writing of individual vector elements are actually handled by functions. If `R` doesn’t already know that `y` is a vector, these functions have nothing on which to act.

Speaking of binding, just as variables are not declared, they are not constrained in terms of mode. The following sequence of events is perfectly valid:

```r
> x <- c(1,5)
> x
[1] 1 5
> x <- "abc"
```

First, `x` is associated with a numeric vector, then with a string. (Again, for C/C++ programmers: `x` is nothing more than a pointer, which can point to different types of objects at different times.)

### 2.3 Recycling

When applying an operation to two vectors that requires them to be the same length, `R` automatically *recycles*, or repeats, the shorter one, until it is long enough to match the longer one. Here is an example:

```r
> c(1,2,4) + c(6,0,9,20,22)
[1] 7 2 13 21 24
```

Warning message:

longer object length
is not a multiple of shorter object length in: c(1, 2, 4) + c(6, 0, 9, 20, 22)

The shorter vector was recycled, so the operation was taken to be as follows:

```r
> c(1,2,4,1,2) + c(6,0,9,20,22)
```
Here’s a more subtle example:

```r
> x
  [,1] [,2]
 [1,]   4
 [2,]   5
 [3,]   6
> x+c(1,2)
  [,1] [,2]
 [1,]   6
 [2,]   7
 [3,]   8
```

Again, keep in mind that matrices are actually long vectors. Here, `x`, as a 3-by-2 matrix, is also a six-element vector, which in R is stored column by column. In other words, in terms of storage, `x` is the same as `c(1,2,3,4,5,6)`. We added a two-element vector to this six-element one, so our added vector needed to be repeated twice to make six elements. In other words, we were essentially doing this:

```r
x + c(1,2,1,2,1,2)
```

Not only that, but `c(1,2,1,2,1,2)` was also changed from a vector to a matrix having the same shape as `x` before the addition took place:

```
1 2
2 1
```

Thus, the net result was to compute the following:

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

### 2.4 Common Vector Operations

Now let’s look at some common operations related to vectors. We’ll cover arithmetic and logical operations, vector indexing, and some useful ways to create vectors. Then we’ll look at two extended examples of using these operations.

#### 2.4.1 Vector Arithmetic and Logical Operations

Remember that R is a functional language. Every operator, including `+` in the following example, is actually a function.
> 2+3
[1] 5
> "+"(2,3)
[1] 5

Recall further that scalars are actually one-element vectors. So, we can add vectors, and the + operation will be applied element-wise.

> x <- c(1,2,4)
> x + c(5,0,-1)
[1] 6 2 3

If you are familiar with linear algebra, you may be surprised at what happens when we multiply two vectors.

> x * c(5,0,-1)
[1]  5  0 -4

But remember, because of the way the * function is applied, the multiplication is done element by element. The first element of the product (5) is the result of the first element of x (1) being multiplied by the first element of c(5,0,-1) (5), and so on.

The same principle applies to other numeric operators. Here’s an example:

> x <- c(1,2,4)
> x / c(5,4,-1)
[1] 0.2 0.5 -4.0
> x %% c(5,4,-1)
[1] 1 2 0

2.4.2 Vector Indexing

One of the most important and frequently used operations in R is that of indexing vectors, in which we form a subvector by picking elements of the given vector for specific indices. The format is vector1[vector2], with the result that we select those elements of vector1 whose indices are given in vector2.

> y <- c(1.2,3.9,0.4,0.12)
> y[c(1,3)]  # extract elements 1 and 3 of y
[1] 1.2 0.4
> y[2:3]
[1] 3.9 0.4
> y[4] <- 3.4
> y[1]
[1] 1.40 0.12
Note that duplicates are allowed.

```r
> x <- c(4,2,17,5)
> y <- x[c(1,1,3)]
> y
[1] 4 4 17
```

Negative subscripts mean that we want to exclude the given elements in our output.

```r
> z <- c(5,12,13)
> z[-1]  # exclude element 1
[1] 12 13
> z[-1:-2]  # exclude elements 1 through 2
[1] 13
```

In such contexts, it is often useful to use the `length()` function. For instance, suppose we wish to pick up all elements of a vector `z` except for the last. The following code will do just that:

```r
> z <- c(5,12,13)
> z[1:(length(z)-1)]
[1] 5 12
```

Or more simply:

```r
> z[-length(z)]
[1] 5 12
```

This is more general than using `z[1:2]`. Our program may need to work for more than just vectors of length 2, and the second approach would give us that generality.

### 2.4.3 Generating Useful Vectors with the : Operator

There are a few R operators that are especially useful for creating vectors. Let’s start with the colon operator `:`, which was introduced in Chapter 1. It produces a vector consisting of a range of numbers.

```r
> 5:8
[1] 5 6 7 8
> 5:1
[1] 5 4 3 2 1
```
You may recall that it was used earlier in this chapter in a loop context, as follows:

```r
for (i in 1:length(x)) {

Beware of operator precedence issues.
```

```r
> i <- 2
> 1:i-1  # this means (1:i) - 1, not i:(i-1)
[1] 0 1
> 1:(i-1)
[1] 1
```

In the expression 1:i-1, the colon operator takes precedence over the subtraction. So, the expression 1:i is evaluated first, returning 1:2. R then subtracts 1 from that expression. That means subtracting a one-element vector from a two-element one, which is done via recycling. The one-element vector (1) will be extended to (1,1) to be of compatible length with 1:2. Element-wise subtraction then yields the vector (0,1).

In the expression 1:(i-1), on the other hand, the parentheses have higher precedence than the colon. Thus, 1 is subtracted from i, resulting in 1:1, as seen in the preceding example.

**NOTE**

You can obtain complete details of operator precedence in R through the included help. Just type `?Syntax at the command prompt.``

### 2.4.4 Generating Vector Sequences with `seq()`

A generalization of : is the `seq()` (or sequence) function, which generates a sequence in arithmetic progression. For instance, whereas `3:8` yields the vector (3,4,5,6,7,8), with the elements spaced one unit apart (4 – 3 = 1, 5 – 4 = 1, and so on), we can make them, say, three units apart, as follows:

```r
> seq(from=12, to=30, by=3)
[1] 12 15 18 21 24 27 30
```

The spacing can be a noninteger value, too, say 0.1.

```r
> seq(from=1.1, to=2, length=10)
[1] 1.1 1.2 1.3 1.4 1.5 1.6 1.7 1.8 1.9 2.0
```

One handy use for `seq()` is to deal with the empty-vector problem we mentioned earlier in Section 2.1.2. There, we were dealing with a loop that began with this:

```r
for (i in 1:length(x))
```
If \( x \) is empty, this loop should not have any iterations, but it actually has two, since \( \text{length}(x) \) evaluates to \( (1,0) \). We could fix this by writing the statement as follows:

```r
def (i in seq(x))
```

To see why this works, let's do a quick test of `seq()`:

```r
> x <- c(5,12,13)
> x
[1] 5 12 13
> seq(x)
[1] 1 2 3
> x <- NULL
> x
NULL
> seq(x)
integer(0)
```

You can see that `seq(x)` gives us the same result as `1:length(x)` if \( x \) is not empty, but it correctly evaluates to `NULL` if \( x \) is empty, resulting in zero iterations in the above loop.

### 2.4.5 Repeating Vector Constants with `rep()`

The `rep()` (or `repeat`) function allows us to conveniently put the same constant into long vectors. The call form is `rep(x, times)`, which creates a vector of `times * length(x)` elements—that is, `times` copies of `x`. Here is an example:

```r
> x <- rep(8,4)
> x
[1] 8 8 8 8
> rep(c(5,12,13),3)
[1] 5 12 13 5 12 13 5 12 13
> rep(1:3,2)
[1] 1 2 3 1 2 3
```

There is also a named argument `each`, with very different behavior, which interleaves the copies of `x`.

```r
> rep(c(5,12,13),each=2)
[1] 5 5 12 12 13 13
```
2.5 Using all() and any()

The any() and all() functions are handy shortcuts. They report whether any or all of their arguments are TRUE.

```r
> x <- 1:10
> any(x > 8)
[1] TRUE
> any(x > 88)
[1] FALSE
> all(x > 88)
[1] FALSE
> all(x > 0)
[1] TRUE
```

For example, suppose that R executes the following:

```r
> any(x > 8)
```

It first evaluates x > 8, yielding this:

`(FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, TRUE, TRUE)`

The any() function then reports whether any of those values is TRUE. The all() function works similarly and reports if all of the values are TRUE.

2.5.1 Extended Example: Finding Runs of Consecutive Ones

Suppose that we are interested in finding runs of consecutive 1s in vectors that consist just of 1s and 0s. In the vector `(1,0,0,1,1,0,1,1)`, for instance, there is a run of length 3 starting at index 4, and runs of length 2 beginning at indices 4, 5, and 8. So the call `findruns(c(1,0,0,1,1,0,1,1),2)` to our function to be shown below returns `(4,5,8)`. Here is the code:

```r
findruns <- function(x,k) {
  n <- length(x)
  runs <- NULL
  for (i in 1:(n-k+1)) {
    if (all(x[i:(i+k-1)]==1)) runs <- c(runs,i)
  }
  return(runs)
}
```
In line 5, we need to determine whether all of the \( k \) values starting at \( x[i] \)—that is, all of the values in \( x[i], x[i+1], \ldots, x[i+k-1] \)—are 1s. The expression \( x[i:(i+k-1)] \) gives us this range in \( x \), and then applying `all()` tells us whether there is a run there.

Let's test it:

```r
> y <- c(1,0,0,1,1,0,1,1)
> findruns(y,3)
[1] 4
> findruns(y,2)
[1] 4 5 8
> findruns(y,6)
NULL
```

Although the use of `all()` is good in the preceding code, the buildup of the vector `run` is not so good. Vector allocation is time consuming. Each execution of the following slows down our code, as it allocates a new vector in the call `c(runs,i)`. (The fact that new vector is assigned to `runs` is irrelevant; we still have done a vector memory space allocation.)

```r
runs <- c(runs,i)
```

In a short loop, this probably will be no problem, but when application performance is an issue, there are better ways.

One alternative is to preallocate the memory space, like this:

```r
findruns1 <- function(x,k) {
  n <- length(x)
  runs <- vector(length=n)
  count <- 0
  for (i in 1:(n-k+1)) {
    if (all(x[i:(i+k-1)]==1)) {
      count <- count + 1
      runs[count] <- i
    }
  }
  if (count > 0) {
    runs <- runs[1:count]
  } else runs <- NULL
  return(runs)
}
```

In line 3, we set up space of a vector of length \( n \). This means we avoid new allocations during execution of the loop. We merely fill `runs`, in line 8. Just before exiting the function, we redefine `runs` in line 12 to remove the unused portion of the vector.
This is better, as we’ve reduced the number of memory allocations to just two, down from possibly many in the first version of the code.

If we really need the speed, we might consider recording this in C, as discussed in Chapter 14.

### 2.5.2 Extended Example: Predicting Discrete-Valued Time Series

Suppose we observe 0- and 1-valued data, one per time period. To make things concrete, say it’s daily weather data: 1 for rain and 0 for no rain. Suppose we wish to predict whether it will rain tomorrow, knowing whether it rained or not in recent days. Specifically, for some number $k$, we will predict tomorrow’s weather based on the weather record of the last $k$ days. We’ll use majority rule: If the number of 1s in the previous $k$ time periods is at least $k/2$, we’ll predict the next value to be 1; otherwise, our prediction is 0. For instance, if $k = 3$ and the data for the last three periods is 1,0,1, we’ll predict the next period to be a 1.

But how should we choose $k$? Clearly, if we choose too small a value, it may give us too small a sample from which to predict. Too large a value will cause us to rely on data from the distant past that may have little or no predictive value.

A common solution to this problem is to take known data, called a training set, and then ask how well various values of $k$ would have performed on that data.

In the weather case, suppose we have 500 days of data and suppose we are considering using $k = 3$. To assess the predictive ability of that value for $k$, we “predict” each day in our data from the previous three days and then compare the predictions with the known values. After doing this throughout our data, we have an error rate for $k = 3$. We do the same for $k = 1$, $k = 2$, $k = 4$, and so on, up to some maximum value of $k$ that we feel is enough. We then use whichever value of $k$ worked best in our training data for future predictions.

So how would we code that in R? Here’s a naïve approach:

```r
preda <- function(x,k) {
  n <- length(x)
  k2 <- k/2
  # the vector pred will contain our predicted values
  pred <- vector(length=n-k)
  for (i in 1:(n-k)) {
    if (sum(x[i:(i+(k-1))]) >= k2) pred[i] <- 1 else pred[i] <- 0
  }
  return(mean(abs(pred-x[(i+1):(n)])))
}
```

The heart of the code is line 7. There, we’re predicting day 1+k (prediction to be stored in pred[i]) from the $k$ days previous to it—that is, days 1, ..., i+k-1. Thus, we need to count the 1s among those days. Since we’re
working with 0 and 1 data, the number of 1s is simply the sum of \( x[j] \) among those days, which we can conveniently obtain as follows:

\[
\text{sum}(x[i:(i+(k-1))])
\]

The use of \texttt{sum()} and vector indexing allow us to do this computation compactly, avoiding the need to write a loop, so it’s simpler and faster. This is typical R.

The same is true for this expression, on line 9:

\[
\text{mean}(\text{abs} (\text{pred}-x[(k+1):n]))
\]

Here, \texttt{pred} contains the predicted values, while \( x[(k+1):n] \) has the actual values for the days in question. Subtracting the second from the first gives us values of either 0, 1, or -1. Here, 1 or -1 correspond to prediction errors in one direction or the other, predicting 0 when the true value was 1 or vice versa. Taking absolute values with \texttt{abs()}, we have 0s and 1s, the latter corresponding to errors.

So we now know where days gave us errors. It remains to calculate the proportion of errors. We do this by applying \texttt{mean()}, where we are exploiting the mathematical fact that the mean of 0 and 1 data is the proportion of 1s. This is a common R trick.

The above coding of our \texttt{preda()} function is fairly straightforward, and it has the advantage of simplicity and compactness. However, it is probably slow. We could try to speed it up by vectorizing the loop, as discussed in Section 2.6. However, that would not address the major obstacle to speed here, which is all of the duplicate computation that the code does. For successive values of \( i \) in the loop, \texttt{sum()} is being called on vectors that differ by only two elements. Except for cases in which \( k \) is very small, this could really slow things down.

So, let’s rewrite the code to take advantage of previous computation. In each iteration of the loop, we will update the previous sum we found, rather than compute the new sum from scratch.

```r
predb <- function(x,k) {
  n <- length(x)
  k2 <- k/2
  pred <- vector(length=n-k)
  sm <- sum(x[1:k])
  if (sm >= k2) pred[1] <- 1 else pred[1] <- 0
  if (n-k > 2) {
    for (i in 2:(n-k)) {
      sm <- sm + x[i+k-1] - x[i-1]
      if (sm >= k2) pred[i] <- 1 else pred[i] <- 0
    }
  }
  return(mean(abs(pred-x[(k+1):n])))
}
```
The key is line 9. Here, we are updating \( s_k \) by subtracting the oldest element making up the sum \( (x[i-1]) \) and adding the new one \( (x[i+k-1]) \).

Yet another approach to this problem is to use the R function \( \text{cumsum()} \), which forms cumulative sums from a vector. Here is an example:

\[
\begin{align*}
  & y <- c(5,2,-3,8) \\
  & \text{cumsum}(y) \\
  & \begin{bmatrix}
  5 & 7 & 4 & 12
  \end{bmatrix}
\end{align*}
\]

Here, the cumulative sums of \( y \) are \( 5, 5 + 2 = 7, 5 + 2 + (-3) = 4, \) and \( 5 + 2 + (-3) + 8 = 12 \), the values returned by \( \text{cumsum()} \).

The expression \( \text{sum}(x[i:(i+k-1)]) \) in \text{pred()}, in the example suggests using differences of \( \text{cumsum()} \) instead:

\[
\begin{align*}
  \text{pred} & \leftarrow \text{function}(x,k) \{ \\
  n & \leftarrow \text{length}(x) \\
  k2 & \leftarrow k/2 \\
  \# \text{ the vector \( \text{red} \) will contain our predicted values} \\
  \text{pred} & \leftarrow \text{vector(length=n-k)} \\
  \text{csx} & \leftarrow c(0,\text{cumsum}(x)) \\
  \text{for} (i \text{ in } 1:(n-k)) \{ \\
    \text{if } (\text{csx}[i+k] - \text{csx}[i] > k2) \text{ pred[i] } \leftarrow 1 \text{ else pred[i] } \leftarrow 0 \\
  \} \\
  \text{return} & \left(\text{mean(abs}(\text{pred}-x[(i+1):n]))\right)
\}
\end{align*}
\]

Instead of applying \( \text{sum()} \) to a window of \( k \) consecutive elements in \( x \), like this:

\[
\text{sum}(x[i:(i+k-1)])
\]

we compute that same sum by finding the difference between the cumulative sums at the end and beginning of that window, like this:

\[
\text{csx}[i+k] - \text{csx}[i]
\]

Note the prepending of a 0 in the vector of cumulative sums:

\[
\text{csx} \leftarrow c(0,\text{cumsum}(x))
\]

This is needed in order to handle the case \( i = 1 \) correctly.

This approach in \text{pred()} requires just one subtraction operation per iteration of the loop, compared to two in \text{predb()}.

### 2.6 Vectorized Operations

Suppose we have a function \( f() \) that we wish to apply to all elements of a vector \( x \). In many cases, we can accomplish this by simply calling \( f() \) on \( x \) itself.
This can really simplify our code and, moreover, give us a dramatic performance increase of hundredsfold or more.

One of the most effective ways to achieve speed in R code is to use operations that are *vectorized*, meaning that a function applied to a vector is actually applied individually to each element.

### 2.6.1 Vector In, Vector Out

You saw examples of vectorized functions earlier in the chapter, with the `+` and `*` operators. Another example is `>`.  

```r
> u <- c(5,2,8)
> v <- c(1,3,9)
> u > v
[1] TRUE FALSE FALSE
```

Here, the `>` function was applied to `u[1]` and `v[1]`, resulting in `TRUE`, then to `u[2]` and `v[2]`, resulting in `FALSE`, and so on.

A key point is that if an R function uses vectorized operations, it, too, is vectorized, thus enabling a potential speedup. Here is an example:

```r
> w <- function(x) return(x+1)
> w(u)
[1] 6 3 9
```

Here, `w()` uses `+`, which is vectorized, so `w()` is vectorized as well. As you can see, there is an unlimited number of vectorized functions, as complex ones are built up from simpler ones.

Note that even the transcendental functions—square roots, logs, trig functions, and so on—are vectorized.

```r
> sqrt(1:9)
[1] 1.000000 1.414214 1.732051 2.000000 2.236068 2.449489 2.645751 2.828427
```

This applies to many other built-in R functions. For instance, let's apply the function for rounding to the nearest integer to an example vector `y`:

```r
> y <- c(1.2,3.9,0.4)
> z <- round(y)
> z
[1] 1 4 0
```

The point is that the `round()` function is applied individually to each element in the vector `y`. And remember that scalars are really single-element vectors, so the "ordinary" use of `round()` on just one number is merely a special case.

---

> round(1.2)
[1] 1

Here, we used the built-in function `round()`, but you can do the same thing with functions that you write yourself:

As mentioned earlier, even operators such as `+` are really functions. For example, consider this code:

```
> y <- c(12,5,13)
> y+4
[1] 16 9 17
```

The reason element-wise addition of 4 works here is that the `+` is actually a function! Here it is explicitly:

```
> '+'(y,4)
[1] 16 9 17
```

Note, too, that recycling played a key role here, with the 4 recycled into (4,4,4).

Since we know that R has no scalars, let’s consider vectorized functions that appear to have scalar arguments.

```
> f
function(x,c) return((x+c)^2)
> f(1:3,0)
[1] 1 4 9
> f(1:3,1)
[1] 4 9 16
```

In our definition of f() here, we clearly intend c to be a scalar, but, of course, it is actually a vector of length 1. Even if we use a single number for c in our call to f(), it will be extended through recycling to a vector for our computation of x+c within f(). So in our call f(1:3,1) in the example, the quantity x+c becomes as follows:

\[
\begin{pmatrix}
1 \\
2 \\
3 \\
\end{pmatrix} + \begin{pmatrix}
1 \\
1 \\
1 \\
\end{pmatrix}
\]

This brings up a question of code safety. There is nothing in f() that keeps us from using an explicit vector for c, such as in this example:

```
> f(1:3,1:3)
[1] 4 16 36
```

You should work through the computation to confirm that (4,16,36) is indeed the expected output.
If you really want to restrict \( c \) to scalars, you should insert some kind of check, say this one:

```r
f <- function(x, c)
  if (length(c) != 1) stop("vector c not allowed")
  return((x+c)^2)
```

### 2.6.2 Vector In, Matrix Out

The vectorized functions we've been working with so far have scalar return values. Calling `sqrt()` on a number gives us a number. If we apply this function to an eight-element vector, we get eight numbers, thus another eight-element vector, as output.

But what if our function itself is vector-valued, as \( \text{z12}() \) is here:

```r
z12 <- function(z) return(c(z,z^2))
```

Applying \( \text{z12}() \) to 5, say, gives us the two-element vector \((5,25)\). If we apply this function to an eight-element vector, it produces 16 numbers:

```r
x <- 1:8
> z12(x)
 [1] 1 2 3 4 5 6 7 8 1 4 9 16 25 36 49 64
```

It might be more natural to have these arranged as an 8-by-2 matrix, which we can do with the matrix function:

```r
> matrix(z12(x),ncol=2)
 [,1] [,2]
[1,]  1  1
[2,]  2  4
[3,]  3  9
[4,]  4 16
[5,]  5 25
[6,]  6 36
[7,]  7 49
[8,]  8 64
```

But we can streamline things using `sapply()` (or `simplify apply`). The call `sapply(x,f)` applies the function \( f() \) to each element of \( x \) and then converts the result to a matrix. Here is an example:

```r
> z12 <- function(z) return(c(z,z^2))
> sapply(1:8,z12)
 [1,]  1  1  1  1  1  1  1  1
 [2,]  2  4  2  4  2  4  2  4
 [3,]  3  9  3  9  3  9  3  9
 [4,]  4 16  4 16  4 16  4 16
 [5,]  5 25  5 25  5 25  5 25
 [6,]  6 36  6 36  6 36  6 36
 [7,]  7 49  7 49  7 49  7 49
 [8,]  8 64  8 64  8 64  8 64
```
We do get a 2-by-8 matrix, not an 8-by-2 one, but it’s just as useful this way. We’ll discuss `apply()` further in Chapter 4.

### 2.7 NA and NULL Values

Readers with a background in other scripting languages may be aware of "no such animal" values, such as None in Python and undefined in Perl. R actually has two such values: NA and NULL.

In statistical data sets, we often encounter missing data, which we represent in R with the value NA. NULL, on the other hand, represents that the value in question simply doesn’t exist, rather than being existent but unknown. Let’s see how this comes into play in concrete terms.

#### 2.7.1 Using NA

In many of R’s statistical functions, we can instruct the function to skip over any missing values, or NAs. Here is an example:

```r
> x <- c(88,NA,12,168,13)
> x
[1] 88   NA  12 168  13
> mean(x)
[1] NA
> mean(x,na.rm=TRUE)
[1] 70.25
> x <- c(88,NULL,12,168,13)
> mean(x)
[1] 70.25
```

In the first call, `mean()` refused to calculate, as one value in `x` was NA. But by setting the optional argument `na.rm` (NA remove) to true (T), we calculated the mean of the remaining elements. But R automatically skipped over the NULL value, which we’ll look at in the next section.

There are multiple NA values, one for each mode:

```r
> x <- c(5,NA,12)
> mode(x[1])
[1] "numeric"
> mode(x[2])
[1] "numeric"
> y <- c("abc","def",NA)
> mode(y[2])
[1] "character"
> mode(y[3])
[1] "character"
```
2.7.2 Using NULL

One use of NULL is to build up vectors in loops, in which each iteration adds another element to the vector. In this simple example, we build up a vector of even numbers:

```
# build up a vector of the even numbers in 1:10
> z <- NULL
> for (i in 1:10) if (i %% 2 == 0) z <- c(z,i)
> z
[1] 2 4 6 8 10
```

Recall from Chapter 1 that %% is the modulo operator, giving remainders upon division. For example, 13 %% 4 is 1, as the remainder of dividing 13 by 4 is 1. (See Section 7.2 for a list of arithmetic and logic operators.) Thus the example loop starts with a NULL vector and then adds the element 2 to it, then 4, and so on.

This is a very artificial example, of course, and there are much better ways to do this particular task. Here are two more ways another way to find even numbers in 1:10:

```
> seq(2,10,2)
[1] 2 4 6 8 10
> 2+1:5
[1] 2 4 6 8 10
```

But the point here is to demonstrate the difference between NA and NULL. If we were to use NA instead of NULL in the preceding example, we would pick up an unwanted NA:

```
> z <- NA
> for (i in 1:10) if (i %% 2 == 0) z <- c(z,i)
> z
[1] NA 2 4 6 8 10
```

NULL values really are counted as nonexistent, as you can see here:

```
> u <- NULL
> length(u)
[1] 0
> v <- NA
> length(v)
[1] 1
```

NULL is a special R object with no mode.
2.8 Filtering

Another feature reflecting the functional language nature of R is filtering. This allows us to extract a vector’s elements that satisfy certain conditions. Filtering is one of the most common operations in R, as statistical analyses often focus on data that satisfies conditions of interest.

2.8.1 Generating Filtering Indices

Let’s start with a simple example:

```r
> z <- c(5,2,-3,8)
> w <- z[z > 8]
> w
[1] 5 8
```

Looking at this code in an intuitive, “What is our intent?” manner, we see that we asked R to extract from z all its elements whose squares were greater than 8 and then assign that subvector to w.

But filtering is such a key operation in R that it’s worthwhile to examine the technical details of how R achieves our intent above. Let’s look at it done piece by piece:

```r
> z <- c(5,2,-3,8)
> z
[1] 5 2 -3 8
> z*z > 8
[1] TRUE FALSE TRUE TRUE
```

Evaluation of the expression `z*z > 8` gives us a vector of Boolean values! It’s very important that you understand exactly how this comes about.

First, in the expression `z*z > 8`, note that `everything` is a vector or vector operator:

- Since z is a vector, that means `z*z` will also be a vector (of the same length as z).
- Due to recycling, the number 8 (or vector of length 1) becomes the vector (8,8,8,8) here.
- The operator `>`, like `+`, is actually a function.

Let’s look at an example of that last point:

```r
> "s"(2,1)
[1] TRUE
> "s"(2,5)
[1] FALSE
```
Thus, the following:

\[ z > 8 \]

is really this:

\[ \sim(z > 8) \]

In other words, we are applying a function to vectors—yet another case of vectorization, no different from the others you’ve seen. And thus the result is a vector—in this case, a vector of Booleans. Then the resulting Boolean values are used to cull out the desired elements of \( z \):

\[
> z[c(\text{TRUE}, \text{FALSE}, \text{TRUE}, \text{TRUE})]
\]

\[ [1] \ 5 \ -3 \ 8 \]

This next example will place things into even sharper focus. Here, we will again define our extraction condition in terms of \( z \), but then we will use the results to extract from another vector, \( y \), instead of extracting from \( z \):

\[
> z \leftarrow c(5,2,-3,8) \\
> j \leftarrow z > 8 \\
> j \\
[1] \ \text{TRUE} \ \text{FALSE} \ \text{TRUE} \ \text{TRUE} \\
> y \leftarrow c(1,2,30,5) \\
> y[j] \\
[1] \ 1 \ 30 \ 5
\]

Or, more compactly, we could write the following:

\[
> z \leftarrow c(5,2,-3,8) \\
> y \leftarrow c(1,2,30,5) \\
> y[z > 8] \\
[1] \ 1 \ 30 \ 5
\]

Again, the point is that in this example, we are using one vector, \( z \), to determine indices to use in filtering another vector, \( y \). In contrast, our earlier example used \( z \) to filter itself.

Here’s another example, this one involving assignment. Say we have a vector \( x \) in which we wish to replace all elements larger than a 3 with a 0. We can do that very compactly—in fact, in just one line:

\[
> x[x > 3] \leftarrow 0
\]
Let’s check:

```
> x <- c(1,3,8,2,20)
> x[x > 3] <- 0
> x
[1] 1 3 0 2 0
```

### 2.8.2 Filtering with the `subset()` Function

Filtering can also be done with the `subset()` function. When applied to vectors, the difference between using this function and ordinary filtering lies in the manner in which NA values are handled.

```
> x <- c(6,1:3,NA,12)
> x
[1]  6  1  2  3  NA 12
> x[x > 5]
[1] 6 NA 12
> subset(x,x > 5)
[1]  6 12
```

When we did ordinary filtering in the previous section, R basically said, “Well, x[5] is unknown, so it’s also unknown whether its square is greater than 5.” But you may not want NAs in your results. When you wish to exclude NA values, using `subset()` saves you the trouble of removing the NA values yourself.

### 2.8.3 The `which()` Function

As you’ve seen, filtering consists of extracting elements of a vector `z` that satisfy a certain condition. In some cases, though, we may just want to find the positions within `z` at which the condition occurs. We can do this using `which()`, as follows:

```
> z <- c(5,2,3,8)
> which(z > 8)
[1] 1 3 4
```

The result says that elements 1, 3, and 4 of `z` have squares greater than 8.

As with filtering, it is important to understand exactly what occurred in the preceding code. The expression

```
z > 8
```

is evaluated to `(TRUE,FALSE,TRUE,TRUE)`. The `which()` function then simply reports which elements of the latter expression are `TRUE`. 

---

http://site.ebrary.com/id/10513550?ppg=73
Copyright © No Starch Press. All rights reserved. May not be reproduced in any form without permission from the publisher, except fair uses permitted under U.S. or applicable copyright law.
One handy (though somewhat wasteful) use of which() is for determining the location within a vector at which the first occurrence of some condition holds. For example, recall our code on page 27 to find the first 1 value within a vector x:

```r
f1 <- function(x) {
  for (i in 1:length(x)) {
    if (x[i] == 1) break  # break out of loop
  }
  return(i)
}
```

Here is an alternative way of coding this task:

```r
f1a <- function(x) return(which(x == 1)[1])
```

The call to which() yields the indices of the 1s in x. These indices will be given in the form of a vector, and we ask for element index 1 in that vector, which is the index of the first 1.

That is much more compact. On the other hand, it’s wasteful, as it actually finds all instances of 1s in x, when we need only the first. So, although it is a vectorized approach and thus possibly faster, if the first 1 comes early in x, this approach may actually be slower.

### 2.9 A Vectorized if-then-else: The ifelse() Function

In addition to the usual if-then-else construct found in most languages, R also includes a vectorized version, the ifelse() function. The form is as follows:

```r
ifelse(b, u, v)
```

where b is a Boolean vector, and u and v are vectors.

The return value is itself a vector; element i is u[i] if b[i] is true, or v[i] if b[i] is false. The concept is pretty abstract, so let’s go right to an example:

```r
> x <- 1:10
> y <- ifelse(x %% 2 == 0, 5, 12)  # %% is the mod operator
> y
[1] 12  5 12  5 12  5 12  5
```

Here, we wish to produce a vector in which there is a 5 wherever x is even or a 12 wherever x is odd. So, the actual argument corresponding to the formal argument b is (F,T,F,T,F,T,F,T,F). The second actual argument, 5, corresponding to u, is treated as (5,5,...)(ten 5s) by recycling. The third argument, 12, is also recycled, to (12,12,...).
Here is another example:

```
> x <- c(5, 2, 9, 12)
> ifelse(x > 6, 2*x, 3*x)
[1] 15 6 18 24
```

We return a vector consisting of the elements of x, either multiplied by 2 or 3, depending on whether the element is greater than 6.

Again, it helps to think through what is really occurring here. The expression \( x > 6 \) is a vector of Booleans. If the \( i^{th} \) component is true, then the \( i^{th} \) element of the return value will be set to the \( i^{th} \) element of \( 2x \); otherwise, it will be set to \( 3x \) \([i]\), and so on.

The advantage of ifelse() over the standard if-then-else construct is that it is vectorized, thus potentially much faster.

### 2.9.1 Extended Example: A Measure of Association

In assessing the statistical relation of two variables, there are many alternatives to the standard correlation measure (Pearson product-moment correlation). Some readers may have heard of the Spearman rank correlation, for example. These alternative measures have various motivations, such as robustness to outliers, which are extreme and possibly erroneous data items.

Here, let's propose a new such measure, not necessarily for novel statistical merits (actually it is related to one in broad use, Kendall's \( \tau \), but to illustrate some of the R programming techniques introduced in this chapter, especially ifelse()).

Consider vectors \( x \) and \( y \), which are time series, say for measurements of air temperature and pressure collected once each hour. We'll define our measure of association between them to be the fraction of the time \( x \) and \( y \) increase or decrease together—that is, the proportion of \( 1 \) for which \( y[i+1]-y[i] \) has the same sign as \( x[i+1]-x[i] \). Here is the code:

```
# findud() converts vector v to 1s, 0s, representing an element
# increasing or not, relative to the previous one; output length is 1
# less than input
1: findud <- function(v) {
2:   vud <- v[-1] - v[-length(v)]
3:   return(ifelse(vud > 0, 1, -1))
4: }
5:
6: uddcor <- function(x,y) {
7:   ud <- lapply(list(x,y),findud)
8:   return(mean(ud[[1]] == ud[[2]]))
9: }
10:
11:
12:
```

```
> # Examples:
> x <- c(5, 2, 9, 12)
> y <- c(1, 3, 5, 7)
> uddcor(x, y)
[1] 0.75
```

Above we show the lower bound, 0, when the two time series are perfectly correlated. A vector with all elements equal to \( 1 \) or \( 0 \) would produce a correlation of 1.
Here's an example:

```r
> x
[1]  5  6 13  3  6  0  1 15 16  8 88
> y
[1]  4  2  3 23  6 10 11 12  6  3  2
> udcart(x,y)
[1] 0.4
```

In this example, x and y increased together in 3 of the 10 opportunities (the first time being the increases from 12 to 13 and 2 to 3) and decreased together once. That gives an association measure of 3/10 = 0.3.

Let's see how this works. The first order of business is to recode x and y to sequences of 1s and –1s, with a value of 1 meaning an increase of the current observation over the last. We've done that in lines 5 and 6.

For example, think what happens in line 5 when we call `findud()` with y having a length of, say, 16 elements. Then v[-1] will be a vector of 15 elements, starting with the second element in v. Similarly, v[-length(v)] will again be a vector of 15 elements, this time starting from the first element in v. The result is that we are subtracting the original series from the series obtained by shifting rightward by one time period. The difference gives us the sequence of increase/decrease statuses for each time period—exactly what we need.

We then need to change those differences to 1 and –1s, according to whether a difference is positive or negative. The `ifelse()` call does this easily, compactly, and with smaller execution time than a loop version of the code would have.

We could have then written two calls to `findud()`: one for x and the other for y. But by putting x and y into a list and then using `lapply()`, we can do this without duplicating code. If we were applying the same operation to many vectors instead of only two, especially in the case of a variable number of vectors, using `lapply()` like this would be a big help in compacting and clarifying the code, and it might be slightly faster as well.

We then find the fraction of matches, as follows:

```r
return(mean(ud[1] == ud[2]))
```

Note that `lapply()` returns a list. The components are our 1/–1-coded vectors. The expression `ud[1] == ud[2]` returns a vector of TRUE and FALSE values, which are treated as 1 and 0 values by `mean()`. That gives us the desired fraction.

A more advanced version would make use of R’s `diff()` function, which does lag operations for vectors. We might, for instance, compare each element with the element three spots behind it, termed a lag of 3. The default lag value is one time period, just what we need here.
> u
[1] 1 6 7 2 3 5
> diff(u)
[1] 5 1 -5 1 2

Then line 5 in the preceding example would become this:

```r
vad <- diff(d)
```

We can make the code really compact by using another advanced R function, `sign()`, which converts the numbers in its argument vector to 1, 0, or –1, depending on whether they are positive, zero, or negative. Here is an example:

```r
> u
[1] 1 6 7 2 3 5
> diff(u)
[1] 5 1 -5 1 2
> sign(diff(u))
[1] 1 1 -1 1 1
```

Using `sign()` then allows us to turn this `udcrr()` function into a one-liner, as follows:

```r
> udcrr <- function(x,y) mean(sign(diff(x)) == sign(diff(y)))
```

This is certainly a lot shorter than the original version. But is it better? For most people, it probably would take longer to write. And although the code is short, it is arguably harder to understand.

All R programmers must find their own “happy medium” in trading brevity for clarity.

### 2.9.2 Extended Example: Recoding an Abalone Data Set

Due to the vector nature of the arguments, you can nest `ifelse()` operations. In the following example, which involves an abalone data set, gender is coded as M, F, or I (for infant). We wish to recode those characters as 1, 2, or 3. The real data set consists of more than 4,000 observations, but for our example, we’ll say we have just a few, stored in `g`:

```r
> g
[1] "M" "F" "F" "I" "M" "M" "F"
> ifelse(g == "M",1,ifelse(g == "F",2,3))
[1] 1 2 2 3 1 1 2
```
What actually happens in that nested ifelse()? Let's take a careful look. First, for the sake of concreteness, let's find what the formal argument names are in the function ifelse():

```r
> args(ifelse)
function (test, yes, no)
```

Remember, for each element of test that is true, the function evaluates to the corresponding element in yes. Similarly, if test[1] is false, the function evaluates to no[1]. All values so generated are returned together in a vector.

In our case here, R will execute the outer ifelse() call first, in which test is g == "M", and yes is 1 (recycled); no will (later) be the result of executing ifelse(g=="F",2,3). Now since test[1] is true, we generate yes[1], which is 1. So, the first element of the return value of our outer call will be 1.

Next R will evaluate test[2]. That is false, so R needs to find no[2]. R now needs to execute the inner ifelse() call. It hasn't done so before, because it hasn't needed it until now. R uses the principle of lazy evaluation, meaning that an expression is not computed until it is needed.

R will now evaluate ifelse(g=="F",2,3), yielding (3,2,2,3,3,3,2); this is no for the outer ifelse() call, so the latter's second return element will be the second element of (3,2,2,3,3,3,2), which is 2.

When the outer ifelse() call gets to test[4], it will see that value to be false and thus will return no[4]. Since R had already computed no, it has the value needed, which is 3.

Remember that the vectors involved could be columns in matrices, which is a very common scenario. Say our abalone data is stored in the matrix ab, with gender in the first column. Then if we wish to recode as in the preceding example, we could do it this way:

```r
> ab[,1] <- ifelse(ab[,1] == "M",1,ifelse(ab[,1] == "F",2,3))
```

Suppose we wish to form subgroups according to gender. We could use which() to find the element numbers corresponding to M, F, and I:

```r
> m <- which(g == "M")
> f <- which(g == "F")
> i <- which(g == "I")
> m
[1] 1 5 6
> f
[1] 2 3 7
> i
[1] 4
```
Going one step further, we could save these groups in a list, like this:

```r
> grps <- list()
> for (gen in c("M","F","I")) grps[[gen]] <- which(g==gen)
> grps

[1] 1 5 6

[2] 2 3 7

[3] 1 4
```

Note that we take advantage of the fact that R’s `for()` loop has the ability to loop through a vector of strings. (You’ll see a more efficient approach in Section 4.4.)

We might use our recoded data to draw some graphs, exploring the various variables in the abalone data set. Let’s summarize the nature of the variables by adding the following header to the file:

```
Gender,Length,Diameter,Height,WholeWt,ShuckedWt,ViscWt,ShellWt,Rings
```

We could, for instance, plot diameter versus length, with a separate plot for males and females, using the following code:

```r
aba <- read.csv("abalone.data",header=T,as.is=T)
grps <- list()
for (gen in c("M","F")) grps[[gen]] <- which(aba==gen)
abam <- aba[grps$M,]
abaf <- aba[grps$F,]
plot(aba$m$length,aba$m$diameter)
plot(aba$f$length,aba$f$diameter,pch="x",new=FALSE)
```

First, we read in the data set, assigning it to the variable `aba` (to remind us that it’s abalone data). The call to `read.csv()` is similar to the `read.table()` call we used in Chapter 1, as we’ll discuss in Chapters 6 and 10. We then form `abam` and `abaf`, the submatrices of `aba` corresponding to males and females, respectively.

Next, we create the plots. The first call does a scatter plot of diameter against length for the males. The second call is for the females. Since we want this plot to be superimposed on the same graph as the males, we set the argument `new=FALSE`, instructing R to not create a new graph. The argument `pch="x"` means that we want the plot characters for the female graph to consist of `x` characters, rather than the default `o` characters.

The graph (for the entire data set) is shown in Figure 2.1. By the way, it is not completely satisfactory. Apparently, there is such a strong correlation between diameter and length that the points densely fill up a section of the
graph, and the male and female plots pretty much coincide. (It does appear that males have more variability, though.) This is a common issue in statistical graphics. A finer graphical analysis may be more illuminating, but at least here we see evidence of the strong correlation and that the relation does not vary much across genders.

![Graph of abalone diameter vs. length by gender](image)

Figure 2.1: Abalone diameter vs. length by gender

We can compact the plotting code in the previous example by yet another use of ifelse. This exploits the fact that the plot parameter pch is allowed to be a vector rather than a single character. In other words, R allows us to specify a different plot character for each point.

```r
pchvec <- ifelse(ab$aGender == "M","o","x")
plot(ab$aLength,ab$aDiameter,pch=pchvec)
```

(Here, we’ve omitted the recoding to 1, 2, and 3, but you may wish to retain it for various reasons.)

## 2.10 Testing Vector Equality

Suppose we wish to test whether two vectors are equal. The naive approach, using `==`, won’t work.

```r
> x <- 1:3
> y <- c(1,3,4)
> x == y
[1] TRUE FALSE FALSE
```

---


Copyright © No Starch Press. All rights reserved. May not be reproduced in any form without permission from the publisher, except fair uses permitted under U.S. or applicable copyright law.
What happened? The key point is that we are dealing with vectorization. Just like almost anything else in R, == is a function.

```r
> "=="(3,2)
[1] FALSE
> i <- 2
> 

In fact, == is a vectorized function. The expression x == y applies the function `==()` to the elements of x and y, yielding a vector of Boolean values.

What can be done instead? One option is to work with the vectorized nature of ==, applying the function all():

```r
> x <- 1:3
> y <- c(1,3,4)
> x == y
[1] TRUE FALSE FALSE
> all(x == y)
[1] FALSE
```

Applying all() to the result of == asks whether all of the elements of the latter are true, which is the same as asking whether x and y are identical.

Or even better, we can simply use the `identical` function, like this:

```r
> identical(x,y)
[1] FALSE
```

Be careful, though because the word `identical` really means what it says. Consider this little R session:

```r
> x <- 1:2
> y <- c(1,2)
> x
[1] 1 2
> y
[1] 1 2
> identical(x,y)
[1] FALSE
> typeof(x)
[1] "integer"
> typeof(y)
[1] "double"
```

So, : produces integers while c() produces floating-point numbers. Who knew?
2.11 Vector Element Names

The elements of a vector can optionally be given names. For example, say we have a 50-element vector showing the population of each state in the United States. We could name each element according to its state name, such as "Montana" and "New Jersey". This in turn might lead to naming points in plots, and so on.

We can assign or query vector element names via the names() function:

```r
> x <- c(1,2,4)
> names(x)
NULL
> names(x) <- c("a","b","ab")
> names(x)
[1] "a"  "b"  "ab"
> x
a  b  ab
1 2 4
```

We can remove the names from a vector by assigning NULL:

```r
> names(x) <- NULL
> x
[1] 1 2 4
```

We can even reference elements of the vector by name:

```r
> x <- c(1,2,4)
> names(x) <- c("a","b","ab")
> x["b"]
b
2
```

2.12 More on c()

In this section, we’ll discuss a couple of miscellaneous facts related to the concatenate function, c(), that often come in handy.

If the arguments you pass to c() are of differing modes, they will be reduced to a type that is the lowest common denominator, as follows:

```r
> c(5,2,"abc")
[1] "5"  "2"  "abc"
> c(5,2,list(a=1,b=4))
[[1]]
[1] 5
[[2]]
[1] 2
```
$a$
[1] 1

$b$
[1] 4

In the first example, we are mixing integer and character modes, a combination that R chooses to reduce to the latter mode. In the second example, R considers the list mode to be of lower precedence in mixed expressions. We’ll discuss this further in Section 4.3.

You probably will not wish to write code that makes such combinations, but you may encounter code in which this occurs, so it’s important to understand the effect.

Another point to keep in mind is that c() has a flattening effect for vectors, as in this example:

```r
> c(5,2,c(1.5,6))
[1] 5.0 2.0 1.5 6.0
```

Those familiar with other languages, such as Python, may have expected the preceding code to produce a two-level object. That doesn’t occur with R vectors though you can have two-level lists, as you’ll see in Chapter 4.

In the next chapter, we move on to a very important special case of vectors, that of matrices and arrays.